Parallelization of Push-based System for

Molecular Simulation Data Analysis with GPU

by

Iliiazbek Akhmedov

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science in Computer Science
Department of Computer Science and Engineering
College of Engineering
University of South Florida

Major Professor: Yicheng Tu, Ph.D.
Srinivas Katkoori, Ph.D.
Sameer Varma, Ph.D.

Date of Approval:
August 23, 2016

Keywords: data processing, cuda optimization, big data,
streaming

Copyright © 2016, Iliiazbek Akhmedov

DEDICATION

I dedicate my thesis work to my family who have always been supporting me throughout my life. I also dedicate it to my closest friends who have shared my passion and encouraged me to explore and try my ideas in life.

ACKNOWLEDGMENTS

I would like to thank my major professor throughout my master thesis Dr. Yicheng Tu for his guidance and knowledge that I received through all the time I worked with him. I really appreciate his contribution and very happy that I got a chance to work with him. I as well want to thank the committee members for their support and encouragement.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 traditional disk random I/O framework 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. Beside it, indexing will also cause delay during write operations and since most of the queries work with the entire data sets, indexing loses its point.

There is a new approach to this problem – 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 filters 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 (it will be explained more in later sections). 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.

CHAPTER 1: INTRODUCTION

In various sciences simulation systems are very crucial and might be the key factors 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. It is important to underline scale of data being discussed. Molecular simulations represent systems starting from thousands of atoms and more which are tracked by frames as snapshot of specific time and can last for months. Thus, the size of data starts from couple of gigabytes up to terabytes and petabytes.

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 can be additional issues while using regular tools. 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 traditional random access disk I/O framework to read from disk chunks by chunks. Beside 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 is 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 management issues should be considered and possibly incorporated in the process of analytic investigation in order to achieve relevant result". [1–5] Another good example would be data scale of Facebook Inc. Being the largest social network, it has huge data coming in daily [6]. The data warehouse of Facebook is measured to be 300PB in 2014, and the daily growth is 600TB, most of the raw data is logs, which, most of the times, are used for analytics. Analytical queries involve entire data sets that will be handy for any future strategical planning in big companies, being able to observe data behavior and much more.

The primary focus and problem in this paper is scientific data analysis. Examples brought up above are real life problems that are very similar to the scientific problem that we are focusing on in terms of data sizes. 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. These systems are most of the time N-body simulations. 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 as mentioned and illustrated in previous work of the system [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 [7, 8]. 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 [7]. 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. [9]. 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 [10–12] as of reusing loaded data through queries produced [13–15].

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. We will talk about this in detail as well in further sections.

1.1 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]. It gets up to over 100 times of speedup which is quite crucial, if we think about days of computations. 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 for certain simulation analysis where sacrifice on loss of data is unbearable. Beside 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. All the benchmarks will be done in comparison with already push-based system versus push-based system with GPU improvements. We think that it is not fair to compare pull-based against GPU push-based approach, since the goal of the paper is to demonstrate capabilities of GPU approach that can beat simple sequential push-based system.

1.2 Our Approach With Improvement

Reviewing the approach we are coming up with and what it was before helps us analyzing the weak parts and things that we can focus on to eliminate old bottlenecks.

Original non-improvement standard approach with pull-based querying is probably one of the most flexible, because nobody has to think about improvements and complexities, it does it for every chunk of data and pulls it whenever it is needed. As it has already been mentioned, at this point, the biggest challenge is that supposing we have M queries, for N atoms we would have to load every chunk M times for N chunks of data (frames), and apparently it is very time consuming of I/O and non-linear computations, and that is only for linear functions. Improvements could be done on caching level, but again the worst case would be the same complexity, because most of the time there is nothing really to cache, and if there is, it is too big to store it in memory, so, it just has to be re-read from file. Using accumulative data structures and trying to optimize by means of them doesn't help either, because, first of all, it is too big data to store in memory, and, secondly, most of the time it is based on data distribution expectation of some kind, which is not the best solution, because the experiments can be different, thus data distribution is sometimes not predictable (boiling down to a specific physics problem will destroy the abstractness and applicability of the system to variety of other problems and will narrow it down making it less useful). In summary, the bottlenecks in this approach are I/O time consumption and slow data processing (especially for non-linear computation).

After all, push-based system for solution of the given problem is tremendously fast in comparison with pull-based system. General idea is to push a chunk of data (frame) through the pre-defined network of queries that will be needed to run on the data sets, which basically means chunk of data will be read from file and loaded into memory only once in order to go through all the queries. As you can observe [1], it can speed up for more than 100 times (depending on workload, number of atoms, and functions being used). It is clear that the biggest challenges in this approach were related to implementation of non-linear functions like SDH, since once the data is pushed through the queries, it is almost not coming back (if it is coming back, the point of optimization is lost), and the idea is to come up with approach to eliminate this overhead and weakness. In this case, it is also important what memory exactly is being used, because trying not to use used parts of the same chunk also matters, because L-level caches of CPU might play a huge role in big data sets. One of the solutions is building quad-tree based data structure that will help optimize the calculation, and it has some expectation from data distribution which can speed up depending on dimensions (2D and 3D). This is a mutual problem of both data loading (memory usage) and its processing, and push-based system is a great approach, because it still remains as an abstract concept applicable to different queries and probably wide variety of other problems of similar kind in different fields.

Finally, we've analyzed some main ideas, advantages, and disadvantages of our approach in this paper which is basically push-based system with optimization on data processing part by means of GPU computation (in other words, parallelism). 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. 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. For example, image processing computations and graphics processing are done on GPUs, because they have a lot of minor primitive independent pixel-based computations.

As it has already been mentioned, 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.

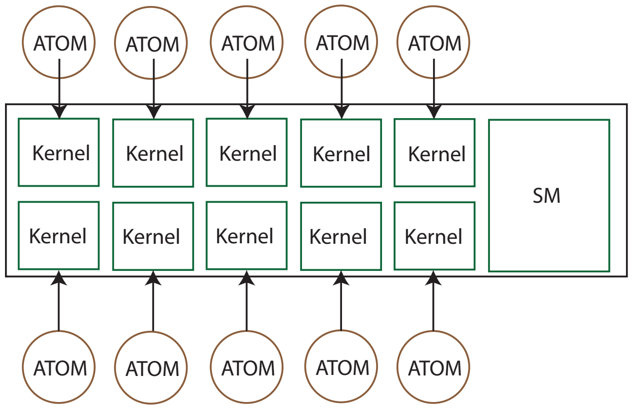


Figure 1.1: Example of atom mapping with kernels

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.

1.3 Contribution And Roadmap Of The Paper

In collaboration with Physics Department, the original motivation came from the need of improving analytic query executions performance over big data being generated by simulation software. The idea was to take input stream coming from simulation software (ideally live) and run network of queries very efficiently, because sometimes they might take days and more to compute. The proposed improved version of push-based system for molecular simulation data analysis, we believe, gives an opportunity for scientists to run their analysis fast having an ability to expand network of queries and customize computation functions to get the desired results. Our system primarily represents a system that works in several modules that were developed in C, CUDA, Python programming languages that, in combination, give us the desired result to perform efficient computations for the network of queries developed in the previous paper [1] as well as to prove the concept of optimization of push-based system by means of parallelism with no data loss:

1. Mock data streamer and generator - tool developed in Python programming language that helps to run performance tests (could also be of other simplest tests like regression) on any kind of data and can be used to simulate random values for physical properties of atoms. The simplest usage of the tool is running a python script with arguments (number of atoms and number of frames).

2. Main execution script - the main module developed in C and CUDA for the actual computation of the network of queries. Network of queries comes predefined and has to be compiled in the script (though later can be abstracted as a configuration file or argument for interpreting general functions), along with computation implementation function that interact with a given GPU.

3. Data conversion - this module comes as a part of the code in Main execution script for the users to be able to manipulate data conversion before loading it to GPU. This is for advanced users having capability of optimizing computations and queries on any level of memory hierarchy (CPU and GPU). It is an abstract part which has to be used upon specific cases to optimize specific functions, since there is no way to predict what functions need to be run and how they can be optimized.

4. Sequential CPU based Push-based system - we re-created original push-based system with the same network of queries as a benchmark for estimation of speedup results of the new approach

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 gets up to 60 times faster, which is just for sampling with no deep improvements on individual queries (since we believe that it should represent average user usage). Original push-based system already had a good speed up in terms of performance in comparison with original pull-based approach and floats around more than 100 times faster depending on number of atoms, frames, and workload. The module for generation of mock data in Python was used for benchmarks, and it represents realistic and pessimistic approach of estimation, since there is no data distribution for physical properties values. We believe that it might be quite important, because experiments may vary widely. 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. This tool can be integrated into open source project of Gromacs as a custom plugin that can be used for optimized analytic queries to be run over data sets live streaming as well as from files. This will be discussed in more detail in further chapters. 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.

CHAPTER 2: RELATED WORK

Push-based System for molecular simulation data analysis with its improved GPU features has couple of main focuses in terms of technical implementation and challenges related to those that have already been tried to be solved by other different approaches and software. Here is the review of other works.

2.1 Scientific Software

First of all, this system is a tool for scientific computation and in this case particularly for molecular simulation data analysis. PeriScope is a project and initiative by University of Michigan which is a system that will optimize data management for scientific explorations. Relational Database Management Systems were created to optimize business applications and it was revolutionary. Those couldn't be built without these systems with abstract languages like SQL, since it manipulates and automatically takes use of indexes, etc. PeriScope is a project that tries to come up with same approach against scientific data and computation with respectively larger scale. [26] It is kind of difficult to compare our proposed system with PeriScope, but the original motivation is very similar: optimize scientific computation. PeriScope is a larger project directed towards more abstract solution like RDBMs that would work for any type of data. Another existing tool for scientific data and optimized processing is SciDB. It was created to make scientific data storage and querying better and faster. It is very similar to traditional RDBMS systems in terms of querying language and represent data storage in tables, but states to be faster than Postgres in 2 orders of magnitude.

Some of the main features is distributed processing with atomicity and durability (the latter is quite difficult for large scale data), while the disadvantages are no-overwriting (only append mode), single value columns only (no arrays, etc), where records are like simple tuples with join function on querying multiple tables [27]. One of the major differences between Push-based Molecular simulation data analysis tool and the projects above is that our proposed design doesn't do anything with storing the data, because we are mainly concentrating it as a side plugin for processing and querying data in an optimized way, while the actual data generation and storage will be done with the actual simulation software.

2.2 Stream Processing

Another major modern issue is live data streaming and live processing, which is also possible with our system. Basically, streaming function is a big challenge and need nowadays, because such networks as Twitter and Facebook are able to keep track of live trends and events, including people's reaction and feedback that have to be reflected real time for tremendous amount of users across the globe. Borealis Stream Processing Software is a tool being developed by MIT, Brown, and Brandeis Universities. Its primary goal is a distributed stream processing and based on Medusa and Aurora. Some of the major functions of the software are highly available distributed large streams processing live, dynamic and highly-scalable with possibility of changing queries live and dynamic vision of query results. [28] There is also a similar need in scientific world, since being able to keep track of results in real time might be crucial to be able to make certain explorations.

As a plugin for optimizing computation of scientific queries against large data sets, an improved Push-based System for MS improvement should be able to handle data streams as well, in order to get real time results and saving up on time, instead of pulling everything from files later on.

For now it has been developed as a separate program that runs independently accepting input in a specified format which is described in the paper, but ideally it is supposed to be embedded right into Gromacs.

2.3 Big Data Frameworks

We are potentially dealing with data that could easily get up to terabytes and petabytes in sizes, and it will be fair to apply term of big data. Traditional software tools are not able to process Big Data apparently because of huge data sets, thus, special frameworks were developed to handle them. Nowadays there are bunch of frameworks that try to solve this problem. Some of these frameworks are Hadoop [29], Hadoop with GPU extension [30], Spark [31], Flink [32] (this is more of a big data stream processing framework) and other ones that are possibly just custom modifications and distributions of these. The mentioned tools help dealing with big data giving an ability to process them in batches on clusters using underlying HDFS (Highly Distributed File System), and normally go along with other utilities that help to apply MapReduce concept. The relationship between our system and this concept along with its implementations is in data sets sizes. Even though we are not using or introducing any non-traditional file systems, in certain scale and problems it could be required, too. In fact, the frameworks listed above could even be used for our problem on top of our system, because, as we discuss in section 6, it could be extended to a distributed system for analyzing independent frames in parallel.

2.4 GPU And Big Data Processing

GPU is becoming more popular being used by not only graphics processing, but machine learning, deep learning, search algorithms, sorting, and so on. Having an ability to optimize in tremendous speedups it is being involved in video classifications, natural language processing.

Particularly about distributed processing and GPU applications along with it, there has been made some overview and research on what is the current state for common solutions in industry. "There were a lot of custom GPU-Hadoop frameworks after the launch of Mars project. These include C-MR, SteamMR, GPMR, Grex, Panda, Shredder and many others. However, most of these frameworks are no longer supported and were developed for some particular scientific projects. Therefore, a Monte Carlo simulation framework can be hardly applied for a bioinformatics project (say) based on the other algorithms. Moreover, processor technologies are evolving very rapidly. Many new revolutionary architectures have been developed for Sony PlayStation 4, Mali GPU by ARM, Adapteva Multicore Microprocessor etc. Both Mali GPU and Adapteva will be compatible with OpenCL. Xeon Phi co-processor works with OpenCL too, which was launched by INTEL. It has an x86-like architecture and is a 60 core co-processor that supports the PCI Express standard. It consumes just 300watts of energy and provides by giving a performance of 1 TFLOPS in double precision. Tianhe-2 which is the most powerful supercomputer till date implements this co-processor. Though, it is very difficult determine which Processor Computing framework architecture will more robust and high performance." [30]

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 [10–12]. 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 [33]. 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 [33]. 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 [28]. On top of the fact that it is a streaming engine, as it was built based on Medusa [34] and Aurora [35], 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.

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 [29]. 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 [30]. 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.

CHAPTER 3: BENCHMARK SCIENTIFIC 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 that were already introduced in the previous work of Push-Based System for Molecular Simulation Data Analysis [1].

In previous work, there has already been developed a network of queries widely used by scientists for MS systems. In Table 3.1 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. Respectively, 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.

The queries that we consider are related to computations limited within one frame of a molecular simulation. 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.

3.1 One Body Functions

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..

Table 3.1: Popular analytical queries in MS

Function Name	Equation
Moment of Inertia	$I = \sum_{i=1}^{n} m_i r_i$
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	$D_z = \sum_{i=1}^{n} \frac{D}{z}$
Density Function	Histogram of atom counts
SDH	Histogram of all distances
RDF	$rdf(r) = \frac{SDH(r)}{4 \cdot \pi \cdot r^2 \cdot \sigma_r \cdot \rho}$

As you can see in the given table, most of them are basically based on primitive summation or other basic mathematical operations, in our experiments in further sections we will be running result on single precision values or to be more explicit we use float type in C programming language.

3.2 Multiple Body Functions

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.

For computation of histogram it is not enough to visit every value for each atom once, since we have to do pairwise computations, thus it increases complexity of the problem not only computation-wise, but memory-wise as well, which is a bigger concern in our case, since the overhead will be huge and non-linearly relative to the size of data. For this purpose we have come up with a way to optimize the way we use references to memory for each atom, so that we minimize overhead of pulling data all the way from DRAM (GPU).

There 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 detail in further sections.

CHAPTER 4: 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 4.1.

4.1 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].

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,

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

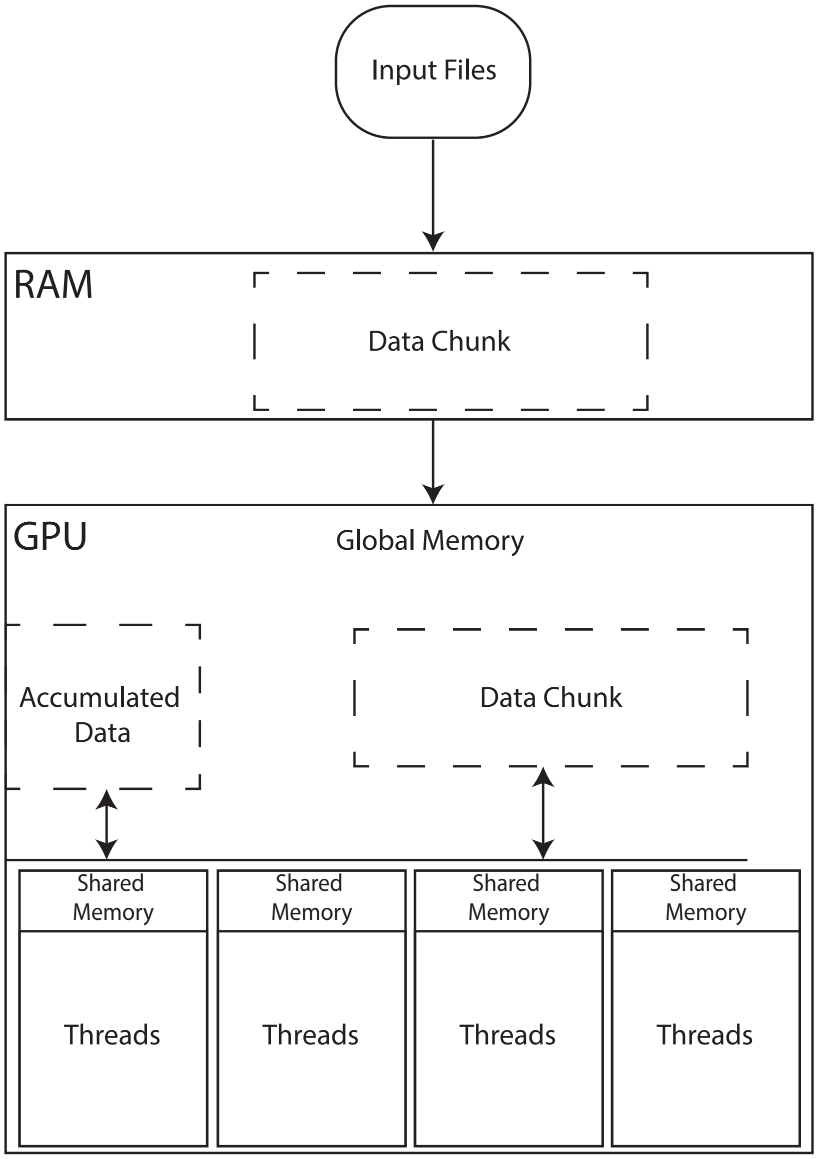


Figure 4.1: Data flow in the improved design of push-based system for MS analysis

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. Potentially input can be technically anything including simple IO sockets that could simply the way the system communicates with other components (f.e. over network) which would also contribute to expanding it to distributed cluster processing (though we believe that it would require additional components such as scheduler or job queues, etc.). To be more exact, the input format of the data and the data represents nothing but a simple array of tuples where each element represents certain descriptor of the given atom in the current frame. We believe that this simple approach makes it a good generalization of solution which would be user friendly for any scientist that wants to run any custom analysis on his data sets.

4.2 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 queries. 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.

Having defined the chunk, we load it from flat format file to RAM. A single atom data

[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

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.

To summarize, the data goes through several levels of memory hierarchy in our system. Coming as an input from either streaming tool, flat file (our case), or other type of input (f.e. socket) which is either disk memory or RAM it goes to RAM of our system process. After it is loaded to RAM, we continue further and load it to Global Memory of GPU, meanwhile while it is loading the given chunk of data to GPU Global Memory (which happens asynchronously), we as well load the next chunk of data to RAM. In this case, it helps us avoid minor overhead of transferring data between CPU to GPU. Further, we take use of different types of memories in GPU to optimize different queries which will be described in further sections.

4.3 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.

From hardware standpoint, it is important to understand that every GPU device consists of SMs (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]. Abstract dimensions given by CUDA are not real entities of physical GPU memory and its organization, but it is just a very convenient way of organizing data and tasks to help GPU scheduler pick and execute the instructions. 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.

Commonly used CUDA array reduction [37] makes the implementation of all the one-body queries obvious. Figure 4.2 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 it could

[3]This explanation doesn't include memory setup in GPU devices, but it is covered in further sections
[4]SIMD - Single Instruction Multiple Data

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.

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.

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

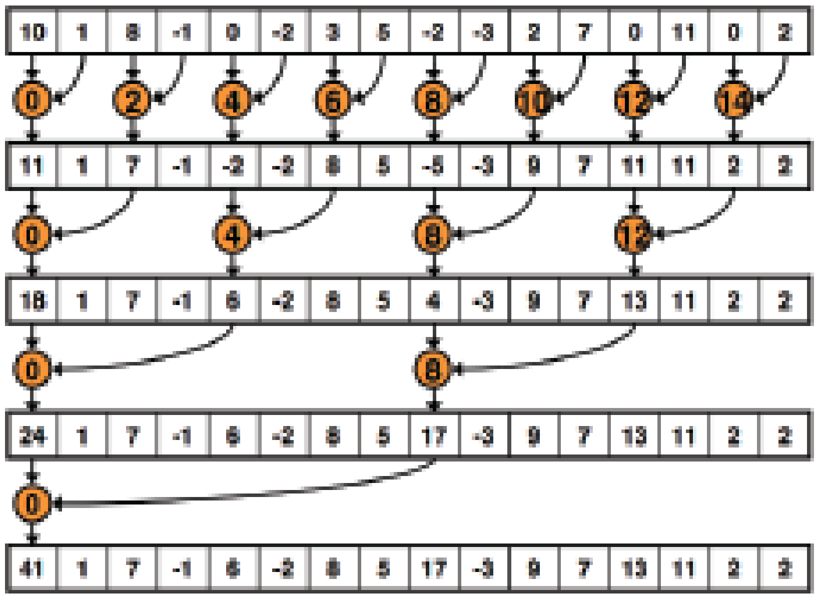


Figure 4.2: GPU array reduction algorithm sample representation

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 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 Figure 4.3). 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

[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

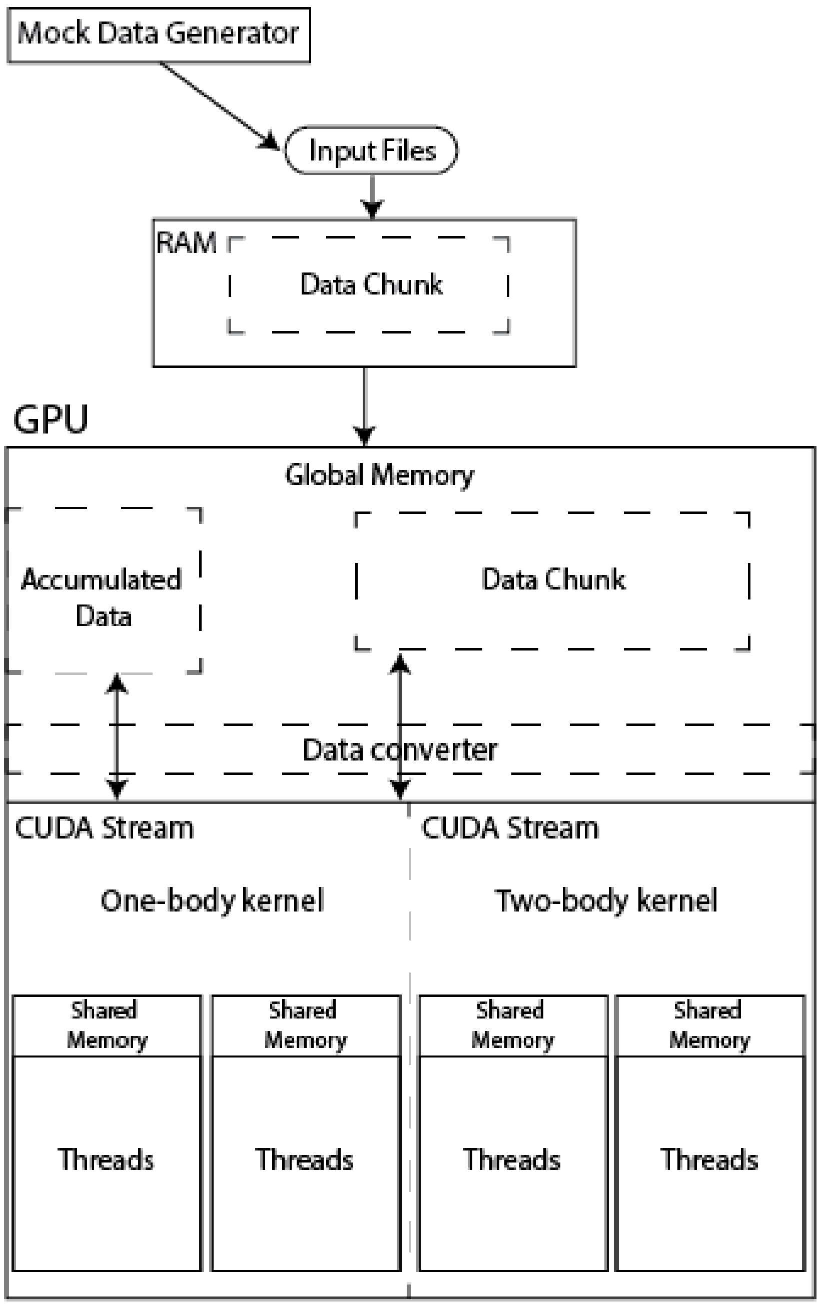


Figure 4.3: Overall system architecture

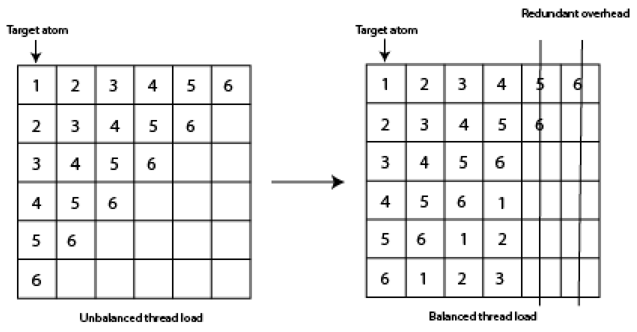


Figure 4.4: SDH thread loading

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 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.

Once the data is loaded into global memory of GPU, we have two kernels for two types of queries setup for two different streams. Beside 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.

As it has already been mentioned in previous sections, one body functions are relatively easy to implement, since they don't require iteration through data more than once with a write approach. Our implementation of one-body functions are relatively easy to understand which is what we tried to explain further.

Basically being given a vector of atoms in CPU version implementation (the one that we use for speedup benchmark) we loop through every atom and accumulate all the needed information to RAM. Once we have the result for one-body functions, we take it out for further usage, it can as well be used for further calculations where there are dependencies on such things as total mass, total charge, etc.

We have provided Pseudocode 1 of the implementation of one body functions for CPU, which has detailed step-by-step description of the CPU sequential version of push-based system for calculating one-body functions.

The approach in GPU version of one-body computation code is a lot different, because there are two major changes: the way we deal with the data in memory and mapping of GPU thread kernels to atoms. Both of these approaches drastically change the way we perceive the solution implementation. Basically in the kernel function we assume we've pre-loaded all the needed atoms for current calculations (loading into memory and higher level parts will be described in later sections).

Algorithm 1 One body functions implementation in CPU version

```
procedure OneBodyFuncsCalc(atoms, atomsCount)
                                                    ▷ $atoms$ is vector of atoms
                                 ▷ $atomsCount$ is number of atoms in the frame
    results ← default initial results value
top:
    i ← 0

    while $i \neq atomsCount$ do                      ▷ Loop through all the atoms
        currentAtom ← atoms[i]

        calcTotalMass(results, currentAtom)
        calcTotalCharge(results, currentAtom)
        calcInertiaX(results, currentAtom)
        calcInertiaY(results, currentAtom)
        calcInertiaZ(results, currentAtom)
        calcDepoleMoment(results, currentAtom)
        // Here we are adding any other one-body functions

        i ← i + 1                                               ▷ Increment i
    end while

    return $results$                                  ▷ Return result for the frame
end procedure

procedure calcTotalMass(results, currentAtom) ▷ Implementation of total mass function
    $results$ ← results.totalmass + currentAtom.mass
end procedure
                                                        ▷ All other functions
```

As you can see from the given algorithm (Pseudocode 2), in the thread kernel function it starts with allocation of memory in the block. To be more explicit, every block has its own level of memory called shared memory. To understand the importance of shared memory, it is on-chip memory which can get latency roughly 100x lower than global memory. Further down, we calculate the current index of the atom we need to work with, and this computation takes two lines of code, because we have a grid-based dimensions that have to be flattened back to one-dimension array indexing which was explained in previous sections. Once we are setup, we start accumulating results into the shared memory atomically, because threads within same block might have race condition on write operation. This is one of the reasons why we are doing it locally, because it would also take much more time and overhead to have locks in global memory that would increase race conditions and locking times. After all, we synchronize all threads to make sure that all the records from atoms have been collected to appropriate local results in shared memory. Finally, we collect all the results from blocks' shared memory into global memory in which case it will be much faster, because number of block is not that big. 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 (we use native CUDA function atomicAdd which guarantees atomicity).

Since one of the physical queries is RDF which has underlying SDH calculation, we will describe our approach with SDH for CPU and GPU versions.

Implementation for CPU version of SDH is very simple, we do 2 nested loops that calculate histogram. Since SDH criteria and parameters might vary by bucket width and box size for the simulation space, we've used one sample size for most of them.

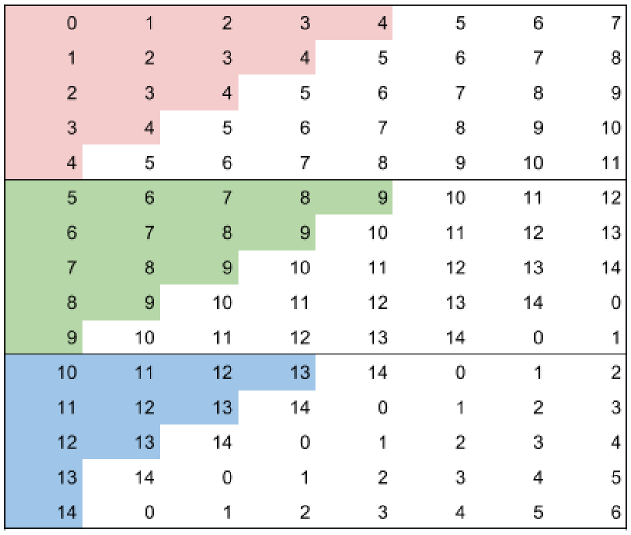
Algorithm 2 Kernel function implementation for one-body queries in GPU

```
procedure GPUONEBODYFUNCTIONSKERNEL(atomsCount, atoms, results)
                                        ▷ atoms is vector of atoms kept in global gpu memory
                    ▷ atomsCount is number of atoms in the frame kept in global gpu memory
                       ▷ results is number of atoms in the frame kept in global gpu memory
top:
    sdata ← allocate shared memory in each GPU block for fast write access
                                ▷ We will use this further to accumulate data across blocks
    indexX ← blockIdsx.x * blockDim.x + threadIdx.x
                        ▷ Identify position of the kernel in the grid scale for coordinate X
    indexY ← blockIdsx.y * blockDim.y + threadIdx.y
                        ▷ Identify position of the kernel in the grid scale for coordinate Y
    gridWidth ← gridDim.x * blockDim.x
                                                      ▷ Calculate grid size in kernel units
    i ← indexY * gridWidth + indexX
                            ▷ Finally get the index of current atom mapped to this kernel
    sharedResult ← pointer to shared memory head
    currentAtom ← atoms[i]

    atomicAdd(&sharedResult[0], currentAtom.mass);

    atomicAdd(&sharedResult[1], currentAtom.charge);
    atomicAdd(&sharedResult[2], currentAtom.mass * currentAtom.x);
    atomicAdd(&sharedResult[3], currentAtom.mass * currentAtom.y);
    atomicAdd(&sharedResult[3], currentAtom.mass * currentAtom.z);
    // other function steps...

    // make sure all kernel threads get here and wait for each other
    __syncthreads();

    // Here we will starting collecting information from each block
    if threadIdx.x = 0 then
        atomicAdd(&results->mass, sharedResult[0]);          ▷ All atomic functions here
        atomicAdd(&results->charge, sharedResult[1]);
        atomicAdd(&results->inertiaX, sharedResult[2]);
        atomicAdd(&results->inertiaY, sharedResult[3]);
        atomicAdd(&results->inertiaZ, sharedResult[4]);
        // other functions to accumulate from block shared memory...
    end if

end procedure
```

Algorithm 3 SDH in CPU version

```
procedure SDHCPU(atoms, atomsCount, histogram)
    $i \leftarrow 0$
    while i < atomsCount do
        $j \leftarrow i + 1$
        while j < atomsCount do
            $dist \leftarrow calcDistance(atoms[i], atoms[j])$
            Save dist in histogram bucket

            $j \leftarrow j + 1$
        end while
        $i \leftarrow i + 1$
    end while
end procedure
```

The approach of solving SDH with GPU is absolutely different just like with one-body queries, though number of computations are the same. We do brute force $O(\frac{N^2}{2})$ complex solution which is pretty much looking at distance between every pair of atoms. In Pseudocode 4 we have tried to provide main logic flow that explains how exactly we implemented kernel for SDH computation. Once the first kernel is finished we run the two-body kernel with its own stream. As it has already been mentioned, the bigger workload is in this kernel, because it has higher complexity. 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 4.4, 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.



0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	0
9	10	11	12	13	14	0	1
10	11	12	13	14	0	1	2
11	12	13	14	0	1	2	3
12	13	14	0	1	2	3	4
13	14	0	1	2	3	4	5
14	0	1	2	3	4	5	6

Figure 4.5: Representation of example of shared memory usage

Here are the main steps of the algorithm:

1. Get mapped atom
2. Load all block related atoms to shared memory
3. Go pairwise with atoms for SDH in a equal balance manner
4. Synchronize threads
5. Get everything for results from shared memory to global memory

As we already mentioned, the main purpose of our proposal is the system that allows integration of GPU into push-based system, thus, we didn't go with too sophisticated and optimized implementations for each query that we benchmark, assuming that users wouldn't do it and having more or less realistic speedup results. Figure 4.5 shows swimlanes as execution thread for blocks. Every row is an execution block for an atom, where first element is the mapped atom. First of all, you can observe that we can clearly see that the execution threads are almost equally balanced (exceptions on odd number of atoms) which means that there is no code divergence and low utilization of kernels (no awaiting redundant threads). In this particular example, we have block size 5 with total number of atoms to be 15 (of course, these are just simple examples, realistically it would be millions). Every swimlane (bordered with black) represent within block execution. Secondly, colored numbers are the atoms that we loaded into shared memory. Of course, this is a small improvement, and we can happily add improvement of adding more shared memory and loading as it moves synchronizing threads in the future.

As we have described in earlier sections, our CPU is primarily used for data flow management, triggering of GPU functions, data structure changes, and GPU computation stream manipulations.

In the Pseudocode 5 you can see rough logic of our CPU mapping function that primarily allocates all the memory in GPU, triggers kernel function and manipulates streams. This is a prototype which definitely is not the last version, because we can as well avoid overhead of copying data into shared memory while synchronously it will be computing data already contained there. The main steps here are the following. We create 2 instances of CUDA streams that are again abstract data types that are scheduled by GPU in a special way. We as well allocate memory in GPU for atoms list, histogram results and common results for one-body queries (in this case it is a data structure with fields that represent some accumulated result). Once we are setup with memory management on GPU, we trigger kernel calls one by one each running on a separate CUDA stream. We don't expect them to run really concurrently, because our two kernels are quite heavily loaded, though streaming can definitely be used for memory management optimization.

Algorithm 4 Kernel function implementation for SDH in GPU

```
procedure GPUSDHKERNEL(atomsCount, pdhRes, histogram, atoms, numOfBuckets)
    smem ← pointer to head of SM
    sharedHisto ← pointer to the head of smem
    sharedAtoms ← pointer to the tail of sharedHisto
    index ← calculate index of kernel mapped to this atom

    if threadIdx.x = 0 then
        sharedHisto ← Set all values to 0
        start ← index, i ← index, k ← 0,
        for $i < start + blockDim.x$ and $i < atomsCount$ do
            sharedAtoms[k] ← atoms[i]            ▷ Load local atoms into shared memory
            i ← i + 1, k ← k + 1
        end for
    end if

    __syncthreads()
    threadLoad = (atomsCount + 1) / 2
    start ← i+1
    end ← i + threadLoad, increment end by one if it is odd and total count is even
    bi ← blockDim.x * blockIdx.x, ei ← bi + blockDim.x
    ind1 ← threadIdx.x
    j ← start
    for j < end do
        ind2 ← j mod atomsCount
        atom1 ← load from shared memory
        if atom2 can be loaded from shared memory then
            atom2 ← load from local memory
        else
            atom2 ← load from global memory
        end if
        calc distance and atomically write to local histogram
    end for

    synchronize all threads
    gather all histogram from shared into global memory
end procedure
```

Algorithm 5 Main algorithm in CPU for managing data and pushing to GPU

```
procedure RUNSINGLEKERNEL(atomsCount, atoms, workload)
    stream1 ← create CUDA stream with cudaStreamCreate
    stream2 ← create CUDA stream with cudaStreamCreate

    gpuResults ← allocate gpu global memory
    gpuAtoms ← allocate gpu global memory
    gpuHistogram ← allocate gpu global memory

    gpuAtoms ← copy atoms from CPU to GPU with cudaMemcpy
    gridSize ← compute grid size depending on atom count

    for $workload$ do
        gpukernel1(stream1, sharedMemorySize, gpuAtoms, ...)      ▷ One body queries kernel
        gpukernel2(stream2, sharedMemorySize, gpuAtoms, ...)      ▷ Two body queries kernel

        cudaStreamSynchronize(stream1)
        cudaStreamSynchronize(stream2)
    end for
end procedure
```

Algorithm 6 Main algorithm in CPU for managing data and pushing to GPU

```
procedure RUNSINGLEKERNEL(atomsCount, atoms, nextAtoms, workload)
    stream1, stream2 ← create CUDA stream with cudaStreamCreate

    The same routing with allocating memory for now for nextAtoms too

    Here we start loading nextAtoms to GPU on stream2

    for $workload$ do
        gpukernel1(stream1, sharedMemorySize, gpuAtoms, ...)      ▷ One body queries kernel
        gpukernel2(stream1, sharedMemorySize, gpuAtoms, ...)      ▷ Two body queries kernel

        cudaStreamSynchronize(stream1)
        cudaStreamSynchronize(stream2)
    end for
end procedure
```

The above Pseudocode 6 shows what memory concurrent could look like. As you can see it is pretty much overlapping $nextAtoms$ loading with computation of $atoms$, which reduces this tiny overhead of copy data over.

CHAPTER 5: EXPERIMENTS RESULTS

The framework has been developed in a programming language CUDA, and the sequential version for benchmark [1] has been originally developed in 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.

5.1 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, Of course, things like PCI bus and all the transportation are important as well, but for speedup benchmarks against CPU vs GPU those stay the same for both sides. RAM on the node that we were running has 16GBs.

We have summarized main CPU parameters for CPU version and it is described in Table 5.1. As you may notice, it is a regular server node CPU with 2.60GHz and 3 levels of L cache.

The same CPU has been used for sequential as well as for improved GPU version during experiments with benchmarks. So, for CPU in GPU improved version refer to the same Table 5.1. For recall, 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.

Table 5.1: Hardware stack for CPU version of Push-based system benchmark

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

Consequently, we also have GPU device with its own performance capacities. Hardware capacities summary for GPU device has been described in Table 5.2

Table 5.2: Hardware stack for GPU version of Push-based system

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

5.2 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 detail 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.

5.3 Results

We have run number of experiments with different number of atoms and frames. Results vary as we choose different data sets, but we can see some tendency towards growth and, fortunately, it is very positive for our improved version of push-based system.

Even though we assume that majority of our cases involve huge amount of atoms (starting from hundred thousands), we still ran experiments on smaller amount of atoms to see difference in speedup relatively with the data set sizes. You can observe summary in Figures 5.1, 5.2, 5.3, 5.4. There is a minor threshold, where CPU version is faster, where workload is less than 20 on Figure 5.1, but it is changed with growth of workload. That presumably is the minor overhead of loading data from CPU to GPU and number of kernels versus computation work actually needs to be done. Once there is more load on computation, strongly GPU accelerates and becomes faster.

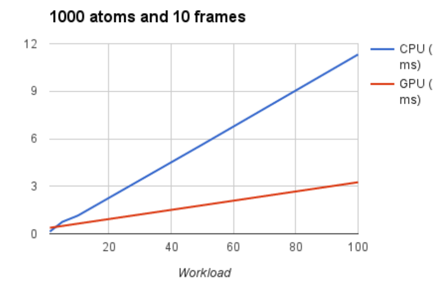


Figure 5.1: Time for 1000 atoms and 10 frames

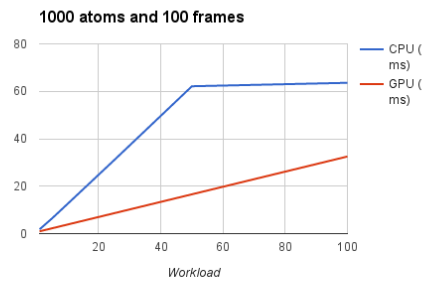


Figure 5.2: Time for 1000 atoms and 100 frames

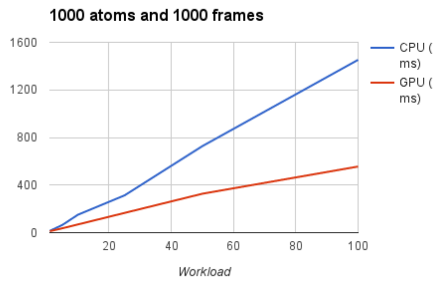


Figure 5.3: Time for 1000 atoms and 1000 frames

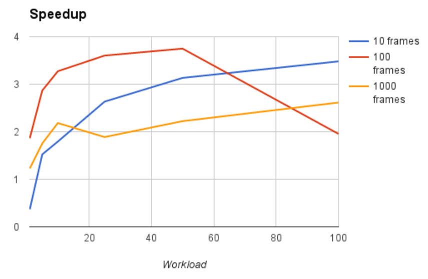


Figure 5.4: Speedup for 1000 atoms

Table 5.3: GPU profiling results for 1000 atoms

Device	Context	Stream	Kernel	Warps Launched	Shared Load
GRID K520 (0)	1	13	$gpu\_one\_body\_function$	32	2
GRID K520 (0)	1	14	$gpu\_two\_body\_function$	32	96080

On Table 5.3 you can observe results of Nvidia Profiling Tool of how much memory call we did to shared memory for SDH computation. You can also notice that there were 32 warps launched for computing 1000 atoms which is absolutely reasonable, since as we already mentioned one warp (scheduling unit for ultimate parallel execution) has 32 threads maximum, thus $\frac{1000}{32} \sim 32$. As expected, stream IDs differ for two kernel executions, and both were run on the same GPU device, though we assume that our system can potentially be extended to using multiple GPU devices as well.

Table 5.4: Memory profiling results for 1000 atoms

$Duration$	$GridSize$	$BlockSize$	$DSMem*$	$Size$	$Throughput$
$1.1830us$	–	–	–	$4B$	$3.3812MB/s$
$6.0470us$	–	–	–	$36.000KB$	$5.9534GB/s$
$1.0240us$	–	–	–	$36B$	$35.156MB/s$
$1.2160us$	–	–	–	$1.4080KB$	$1.1579GB/s$
$234.16us$	$(1,1,1)$	$(1024,1,1)$	$40.960KB$	–	–
$3.1640ms$	$(1,1,1)$	$(1024,1,1)$	$38.272KB$	–	–
$2.5280us$	–	–	–	$36B$	$14.241MB/s$
$2.4960us$	–	–	–	$1.4080KB$	$564.10MB/s$

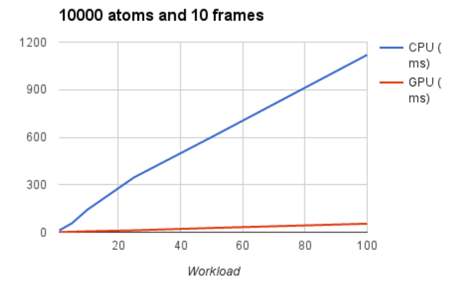


Figure 5.5: Time for 10000 atoms and 10 frames

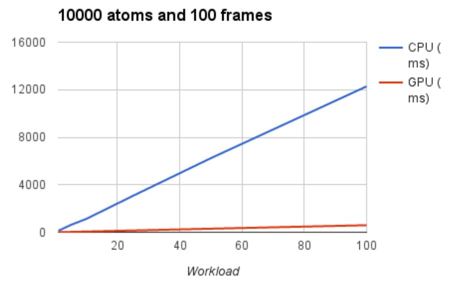


Figure 5.6: Time for 10000 atoms and 100 frames

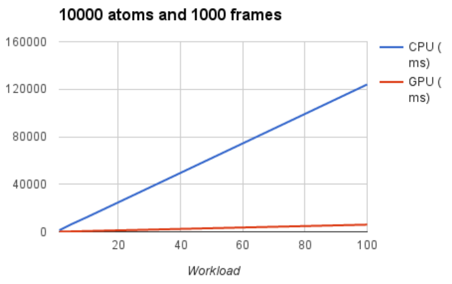


Figure 5.7: Time for 10000 atoms and 1000 frames

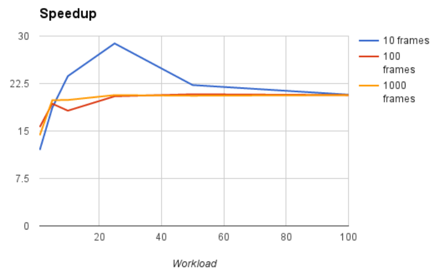


Figure 5.8: Speedup for 10000 atoms

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.

Table 5.5: GPU profiling results for 10,000 atoms

Device	Context	Stream	Kernel	Warps Launched	Shared Load
GRID K520 (0)	1	13	$gpu\_one\_body\_function$	320	20
GRID K520 (0)	1	14	$gpu\_two\_body\_function$	320	5,181,545

Table 5.6: Memory profiling results for 10,000 atoms

$Duration$	$GridSize$	$BlockSize$	$DSMem*$	$Size$	$Throughput$
$1.1840us$	–	–	–	$4B$	$3.3784MB/s$
$42.470us$	–	–	–	$360.00KB$	$8.4766GB/s$
$1.0240us$	–	–	–	$36B$	$35.156MB/s$
$1.2480us$	–	–	–	$1.4080KB$	$1.1282GB/s$
$456.86us$	$(10, 1, 1)$	$(1024, 1, 1)$	$40.960KB$	–	–
$59.627ms$	$(10, 1, 1)$	$(1024, 1, 1)$	$38.272KB$	–	–
$2.7520us$	–	–	–	$36B$	$13.081MB/s$
$2.7850us$	–	–	–	$1.4080KB$	$505.57MB/s$

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

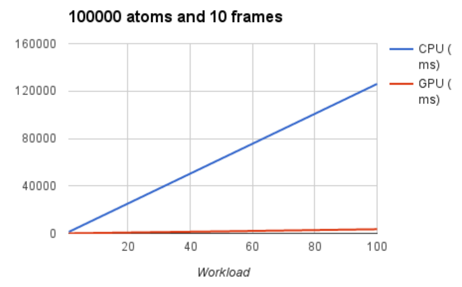


Figure 5.9: Time for 100,000 atoms and 10 frames

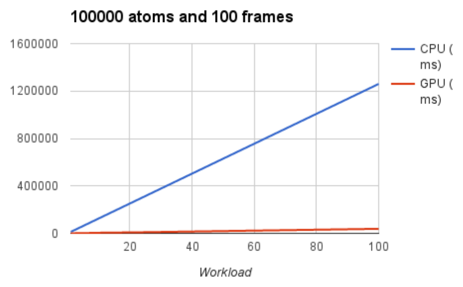


Figure 5.10: Time for 100,000 atoms and 100 frames

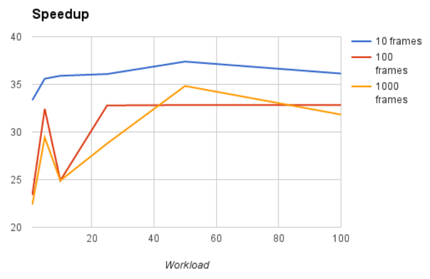


Figure 5.11: Speedup for 100,000 atoms

which is pretty fast, too, while GPU does pretty straightforward multicore processing, which is the main reason it stagnates. Result of another profiling on Table 5.10 changed significantly in terms of warps launched, which is reasonable, and shared memory. It is also noticeable that growth of shared memory loading instructions is non-linear which you can observe in the results for 100K atoms.

When we finally get to the number of atoms that are realistic to the ones scientists use, happily, speedup gets pretty good. Speedup gets up to almost 40 times (Figure 5.11).

Fortunately, scientific data is much bigger than above examples. You can see average speedup for bigger number of atoms on Figure 5.14. The speedup gets up to 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.

Table 5.7: Memory profiling results for 100,000 atoms

$Duration$	$GridSize$	$BlockSize$	$DSMem*$	$Size$	$Throughput$
$1.1840us$	–	–	–	$4B$	$3.3784MB/s$
$729.81us$	–	–	–	$3.6000MB$	$4.9328GB/s$
$992ns$	–	–	–	$36B$	$36.290MB/s$
$1.2160us$	–	–	–	$1.4080KB$	$1.1579GB/s$
$3.0014ms$	$(98, 1, 1)$	$(1024, 1, 1)$	$40.960KB$	–	–
$3.86896s$	$(98, 1, 1)$	$(1024, 1, 1)$	$38.272KB$	–	–
$3.0720us$	–	–	–	$36B$	$11.719MB/s$
$2.9760us$	–	–	–	$1.4080KB$	$473.12MB/s$

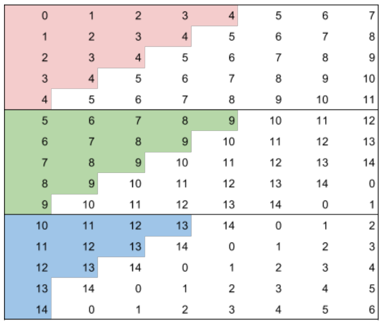
Table 5.8: GPU profiling results for 100,000 atoms

Device	Context	Stream	Kernel	Warps Launched	Shared Load
GRID K520 (0)	1	13	$gpu\_one\_body\_function$	3136	196
GRID K520 (0)	1	14	$gpu\_two\_body\_function$	3136	473,692,099

Finally, for 100,000 atoms the warps launched for execution seems to be growing reasonable along with number of atoms (Table 5.10), while on the other hand usage of shared load for SDH computation is extremely high.

We implemented more advanced version of SDH algorithm taking better use of shared memory just to demonstrate what improvement and speedup could be achieved if we focus on particular query implementation.

Basically, as you can see from Figure 5.12, now we allocate more shared memory. Yellow sections are the atoms that are stored in shared memory throughout the whole thread execution of the entire block, since those atoms are needed at every step, while the green ones are the working zones. We load twice size of the block into shared memory, because the very first one thread in the



0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
2	3	4	5	6	7	8	9
3	4	5	6	7	8	9	10
4	5	6	7	8	9	10	11
5	6	7	8	9	10	11	12
6	7	8	9	10	11	12	13
7	8	9	10	11	12	13	14
8	9	10	11	12	13	14	0
9	10	11	12	13	14	0	1
10	11	12	13	14	0	1	2
11	12	13	14	0	1	2	3
12	13	14	0	1	2	3	4
13	14	0	1	2	3	4	5
14	0	1	2	3	4	5	6

Figure 5.12: New approach of using shared memory

block is going to communicate starting from second one, while the last one will start with the next block, which means that it will have to be twice size. Fortunately, the first part of the threads in the block will consequently use the next block which is already loaded, thus, at each step we load one more block size simply moving window and re-using the ones that are already loaded for first part.

Table 5.9: Memory profiling results for 100,000 atoms in improved SDH

$Duration$	$GridSize$	$BlockSize$	$DSMem*$	$Size$	$Throughput$
$1.4080us$	–	–	–	$4B$	$2.8409MB/s$
$1.0769ms$	–	–	–	$3.6000MB$	$3.3429GB/s$
$1.0880us$	–	–	–	$36B$	$33.088MB/s$
$1.2800us$	–	–	–	$1.4080KB$	$1.1000GB/s$
$3.0340ms$	$(98, 1, 1)$	$(1024, 1, 1)$	$40.960KB$	–	–
$24.5691s$	$(98, 1, 1)$	$(1024, 1, 1)$	$38.272KB$	–	–
$2.7520us$	–	–	–	$36B$	$13.081MB/s$
$2.7830us$	–	–	–	$1.4080KB$	$505.93MB/s$

Table 5.10: GPU profiling results for 100,000 atoms for improved SDH

Device	Context	Stream	Kernel	Warps Launched	Shared Load
GRID K520 (0)	1	13	$gpu\_one\_body\_function$	3136	196
GRID K520 (0)	1	14	$gpu\_two\_body\_function$	3136	246,323,337

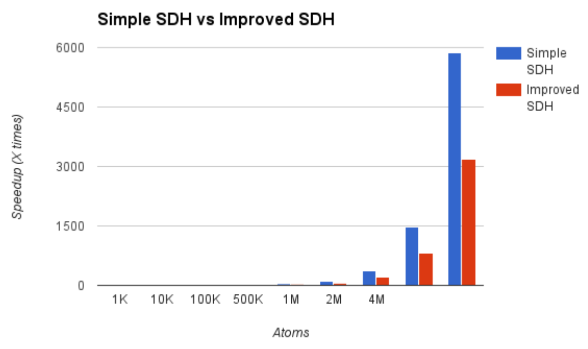


Figure 5.13: Comparison of GPU SDH with GPU SDH improved

As you may have noticed, even though we increased utilization of shared memory, number of loads is around the same. The trick is that we replaced atoms in shared memory with simpler data structure called $coordinates$ that would contains 3 float type variables which saves huge amount of memory. The results that we have achieved are extremely great, in general all the run times get twice less in comparison with simple SDH approach. You may observe run times for some sample data sets on Figure 5.13. In general, it is quite visible that it is twice less. This is a sample improvement which proves that a little more elaboration on a particular query can bring much better results which could end up saving scientists days and months of time of computation.

It is noticable that growth of the speedup is not stable and with certain number of atoms it drops dramatically. One of possible reasons could the fact that we've used static approach of computing grid dimensions. Beside 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.

With improved version of SDH speedup gets up to more than 75 times (Figure 5.14). We believe that these numbers could get even better depending on queries and elaboration of improvement of particular cases.

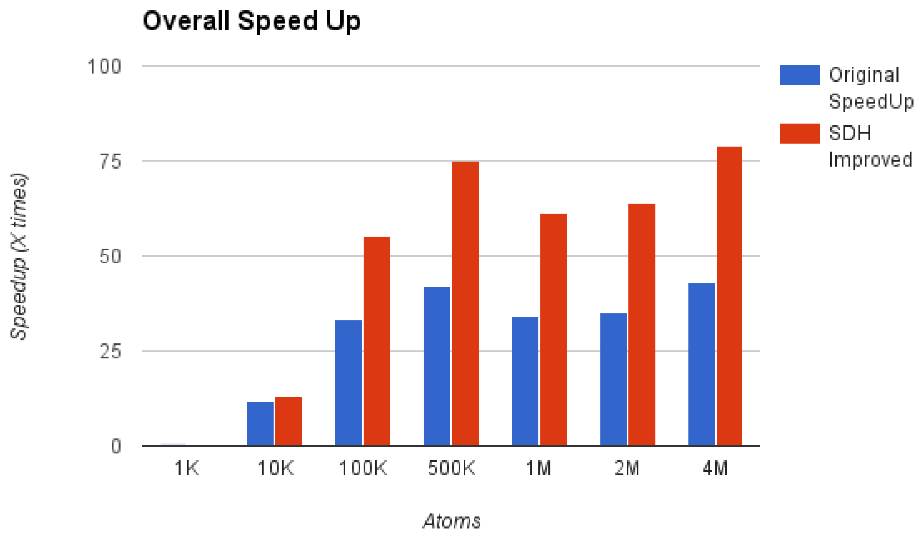


Figure 5.14: Speedup for larger number of atoms

CHAPTER 6: CONCLUSION

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, the proposed improved version of the system could save 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] V. Grupcev, Y.-C. Tu, J. C. Fogarty, and S. Pandit, “Push-based system for molecular simulation data analysis,” in *BIGDATACONF*, 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] Pamela Vagata, *et. al.*, “Scaling the Facebook data warehouse to 300 PB.” [Online]. Available: https://code.facebook.com/posts/229861827208629/scaling-the-facebook-data-warehouse-to-300-pb/

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

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

[9] Gromacs group, “GROMACS - Online Reference.” [Online]. Available: http://gromacs.org/

[10] 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.

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

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

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

[14] 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.

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

[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] Amazon, "Amazon GPU Instances."

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

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

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

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

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

[31] Apache, "Apache Spark." [Online]. Available: http://spark.apache.org/

[32] Flink, "Flink." [Online]. Available: https://flink.apache.org/

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

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

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

[36] *et. al.*. Yicheng Tu, "Computing spatial distance histograms for large scientific datasets on-the-fly."

[37] *et. al.*. Mark Harris, "Optimizing parallel reduction in cuda."