Using State-of-the-Art GPGPU`s for Molecular Simulation:
Optimizing Massively Parallelized N-Body Programs Using NVIDIA Tesla

Romeo C. Buenaflor Jr.
Abstract

Using State-of-the-Art GPGPU's for Molecular Simulation

Romeo C. Buenafior Jr.

Computation and simulation as a tool for discovering new knowledge is still marred by problems that are intractable, combinatoric, or simply plagued by the so called curse of dimensionality. The algorithm used for molecular simulation of polyelectrolytes is one of those areas in computational chemistry and science suffering from the curse of dimensionality. Much of the problems, though, have been claimed to be solvable with the advent of more sophisticated and powerful computers and related technologies. This paper attempts to substantiate the claim. In this paper, a state-of-the-art NVIDIA Tesla C870 has been utilized to massively parallelize the algorithm for the molecular simulation of polyelectrolytes. In particular, this paper attempts to optimize the portion of the code involving the computation of electrostatic interaction using the new technology. It has been shown that tapping this new line of technology poses great advantage in winning the war against the curse of dimensionality.
ACKNOWLEDGMENT

I wish to thank all the people who, in one way or another, have helped me during my two years stay in Uppsala Universitet and Sweden. In particular, I would like to thank: Dr. Malek Khan, my thesis supervisor; Dr. Per Lötstedt, my thesis reviewer; Dr. Anders Jansson, my thesis examiner; and Dr. Stefan Pålsson, my academic adviser, for all their guidance and support.

My heartfelt gratitude to my family, especially my parents.

Romeo C. Buenafior Jr.
Table of Contents

Part I Introduction ................................................................. 1
  1 Background Application of the Problem 1
  2 The Algorithm 1

Part II Hardware ........................................................................ 3
  3 Description 3
  4 Host Computer 3
  5 Architecture of NVIDIA Tesla C870 GPU 5

Part III Operating System .......................................................... 8

Part IV Software Compiler and Tools .......................................... 8
  6 Description and Language 8
  7 Model and Architecture 9

Part V Statement of the Problem and Goals ................... 11

Part VI Scope and Limitation .................................................. 11

Part VII Software Implementation ........................................... 11
  8 Description 11
  9 Software Models 11
 10 Tests 13

Part VIII Experiments .............................................................. 14
  11 Molecular Simulation Phase 1 15
  12 Molecular Simulation Phase 2 15

Part IX Discussion and Analysis ........................................... 15
  Phase 1 15
  Phase 2 27

Part X Conclusion ................................................................. 29

Part XI Recommendation for Future Work ......................... 33

Part XII Summary of NVIDIA GPGPU Cards ................... 36

References .............................................................................. 36
Part I
INTRODUCTION

1 Background Application of the Problem

An ongoing research on polyelectrolyte folding is the source of the problem. The polymer is modelled as follows:

The basic system consists of a single polyelectrolyte with a total number of monomers \( N \). The polyelectrolyte is modelled as a chain of identical hard sphere monomers each carrying a charge \( q_m \). Each monomer is connected to its neighbour by a rigid but freely rotating bond of fixed length \( b \). The polymer is enclosed in a spherical cell, with radius \( R_c \), and the middle monomer is fixed at the centre of the cell. Also in the cell are hard sphere counterions; \( N_c \) is the number of neutralising counterions each with charge \( q_c \). In the cell model, only interactions within the cell are accounted for and no particles are allowed to escape the cell. All simulations are carried out in the so-called Primitive Model whereby the solvent is described by the dielectric constant \( \epsilon \).

The Hamiltonian for the system is the sum

\[
U = U_{el} + U_{hc} + U_{cell}
\]

, where the electrostatic term

\[
U_{el} = \sum_{i<j}^{N+N_c} \frac{q_i q_j e^2}{4\pi\epsilon_0|r_i - r_j|}
\]

, and \( e \) is the elementary charge, \( r_i \) is the coordinate and \( q_i \) is the valence of particle \( i \). \( U_{hc} \) is a hard core potential preventing the particles from getting closer than a distance \( d \) (the hard sphere diameter) from each other, and \( U_{cell} \) is the term that confines all ions and monomers to reside within the spherical cell of radius \( R_c \).

Simulations are carried out in the canonical ensemble and new monomer configurations are generated by pivot moves \([12]\) and the small ions are translated. The chosen monomer divides the polymer in two and the smaller part of the polyelectrolyte is rotated. This ensures that the middle monomer is kept fixed.

In this simulation experiment to determine the best polymer configuration on the basis of energy of the system, it has been found out that the software used spends 80% (and for increasing number of beads, virtually all) of the time extracting the square roots more than in any other part of the program. The function is embedded in the distance calculations between two pairs of molecules or beads in the electrostatic interaction term, \(|r_i - r_j|\) in \([1]\). It is basic that the distance in the three-coordinate-space is given by

\[
|r_i - r_j| = \sqrt{(r_{x,i} - r_{x,j})^2 + (r_{y,i} - r_{y,j})^2 + (r_{z,i} - r_{z,j})^2}
\]

Because the long range forces are significant in this type of simulation, the total number of distance calculations in its rawest form involves an order of \( \mathcal{O}(N^2 - N) \), invoked in as many iterations as required. The problem is basically to reduce the time for these calculations.

2 The Algorithm


\[2\] Detailed results will be included in the supplemental report to be published at a future time.
Figure 1: This is a diagram of the simulation program showing its essential subroutines (tab symbols) and loops (shaded blocks). Note the yellow tab for the fastqin algorithm which is where the program spends virtually all its time. NPass is a user input not less than 100, but in the simulation NPass = 4000. N is the number of beads. Note that the total iterations in the simulation involving the fastqin algorithm is then 4000*(N-1).
To implement the model just described, the original program used for the simulation is written in Fortran. The program can perform simulation for a minimum of 100 molecules or beads to about 2048 maximum. Without actually showing the details of the program, Figure 1 on page 2 and Figure 2 on page 4 illustrate the flow. The portion involved in calculating 1 on page 1 is in the fastqin algorithm. More about this will be discussed below.

Part II
HARDWARE

To implement the time reduction objectives, two hardware platforms on which the software will run have been chosen. One of them is the subject of this thesis.

3 Description

The NVIDIA Tesla C870 GPU, hereafter referred to as GPGPU Tesla, is one of those which can be called an evolved GPU from its predecessors which were all games or graphics oriented cards. This new one shifts its focus to computing applications. As such, it has no display connectors, but still retains full OpenGL and DirectX functionality allowing them to power applications based on these Application Programming Interfaces (APIs) in addition to the Compute Unified Device Architecture (CUDA) Software Developer Kit (SDK). At the outset, the following are its summary specification:

- One GPU with 128 thread processors;
- 518 gigaflops at peak performance;
- 1.5 gigabytes of dedicated memory; and
- fits in one full-length, dual slot with one open PCI Express x16 slot.

For purposes of this paper, two cards have been planned to be utilized with a combined peak performance of one teraflop. As will be discussed, however, only one has been successfully made to run.

4 Host Computer

The cards serve as a high efficiency embedded co-processor of a host computer. For purposes of this paper, the host computer utilized is an HP xw 4600 80+ Energy Efficient Chassis with Intel Core 2 64 Architecture Dual-Core E8400 Processor running at 3.00 GHz. The system has 6MB L2 (shared) cache, 1333 MHz Front Side Bus, equipped with Virtualization Technology. It is also loaded with HP xw 4600 Localization Kit and HP 2GB (2x 1GB) DDR2-800 ECC Memory. Among other things, it mounts an HP 500 GB SATA 3Gb/s NCQ 7200, Int H.

The motherboard has two PCI Express x16 slots, and other slots.

One of the biggest sources of problems in the project is the conflict caused by an old generation graphics card to connect the display monitor, and GPGPU Tesla. The graphics card is of old PCI type. The graphics finally utilized without problem is of PCI Express type.

3 For more detailed information about the card, please read [Cuda Prog Guide] and [PTX]. There are also a lot of information about the card and other related products in the website of NVIDIA. Please see [Computing Tech Brief].
4 See [Computing Tech Brief], p 4.
5 Information is obtained from a leaflet that came with the machine, and the invoice receipt. The author may have also gotten some information from the website of HP.
Figure 2: These are the different diagrams showing the locations of the different subroutines invoked in the main program. Note the different parts of the yellow tab for fastqin. Each shaded block sets up the calculation for the different pairwise distance of the beads, both for the old and the new configuration. Note also the cyan tab for the vector rsqrt operations.
Henceforth, no further treatment of the architecture of the host shall be considered, except when this is relevant to establish the properties of the embedded cards. Please note, however, that other hardware platforms such as the servers available at UPPMAX have been utilized to verify the robustness or accuracy of some portions of the code. These platforms will be discussed together with the tests.

5 Architecture of NVIDIA Tesla C870 GPU

A comprehensive literature for the hardware architecture of the specified card is not available. The only pieces of information are those mixed with the documentation of CUDA. For example, although it has been mentioned that the special processing unit responsible for special functions such as rsqrt is fully pipelined, nowhere in the literature can be found how this pipelining has been implemented. It is not known how deep the pipeline is. While the information is limited and incomplete, the information that follows presents so much that is important to determine and establish the optimization of the algorithms employed by this paper. In fact for the remainder of this subsection, mainly architectural elements which are useful for optimization are featured. A diagram of the architectural elements of the card is presented in Figure 3 on page 6.

5.1 Between Host and Device

As mentioned above, the device is connected to the host via a one full-length, dual slot with one open PCI Express x16 slot. Unfortunately, this is also the part of data transfer where the latency is worst. Although there is no exact figure of this latency, there is a value 4 gigabytes/sec peak bandwidth measured on nForce 680i motherboards using Quadro FX 5600 connected on overclocked PCI-e x16. The peak performance involves the faster page-locked memory transfer facility, which can also degrade the overall performance of the system. The applicability of the figure will have to be verified in this paper.

5.2 Streaming Multiprocessors

The GPGPU Tesla is made up of 16 streaming multiprocessors. Each multiprocessor has access to the global memory. Each has its own local memory shared by the cores. It has also a separate constant cache and texture cache. It likewise has a set of registers that is distributed to the different cores during runtime. There is, however, no available cache for data from the global memory.

5.3 Processor Cores

There are 8 processor cores for each microprocessor, and these can be tapped optimally to run in SIMD fashion. However, for divergent codes inside a conditional which makes different cores execute different tasks, the cores are run serially per divergent code. This is to be avoided as much as possible.

Each core is equipped with its own arithmetic logic unit (ALU) to perform the basic mathematical computations. It has access also to a high speed computing component described next. Through the shared memory, the processor cores within a multiprocessor which belong to the same block of computing units can communicate and synchronize with each other. Outside this block, the two latter features are not available.

5.4 Special Floating Unit (SFU)

There are two SFUs servicing each multiprocessor. These perform calculations that normally take long to perform such as the rsqrt function, sqrt functions and others. It is fully pipelined. This is a possible bottleneck as there are eight cores using each SFU.

---

6See [Cuda Prog Guide], 85.3, p 63.
7See [Cuda Prog Guide], 81.5.1.2, p 30.
Figure 3: The figure illustrates the several essential components of the NVIDIA Tesla C870. Note that two streaming multiprocessors share the set of constant and texture memory.
5.5 Memory

There are different types of memory and registers available with the GPGPU Tesla as already mentioned above. They will be described here in more detail in the order of the latency in which they can be accessed from the slowest to the fastest.

5.5.1 Global Memory

The global memory is the main link to the outside world. There are about 1.5 gigabytes of space available. To access this memory, however, one needs to remember that there are alignment requirements that need to be followed, whether it is a read or a write operation. Data should be aligned in 4, 8, 16, and 32 bytes. A group of cores in half-warp (16 cores) should access the memory such that the size of data per core follows the alignment rule and the chunk of data is aligned in \(16 \times \text{size}\) of each data where the first in the chunk goes to the first thread. Any unaligned data results in split and another round of loading which increases the time. Transfers following both alignment rules are said to be coalesced.

Access to global memory from the host CPU can be done in two ways, pageable and pinned. Pageable memory refers to the ordinary memory available from the host, while pinned or page-locked memory are those specifically assigned to the GPGPU Tesla, which it can have direct access. The access time of the latter is much faster compared to the former. Caution, however, on the use of the pinned memory as too much can slow down the performance of the CPU. In fact, this has been one of the silent problems bugging Phase 2 of the experiments. It has not been discovered for several months. There is virtually no debugger for the CUDA part of the code, and certainly not one to detect pinned memory abuse. The abuse resulted in the virtual halt of the calculation. All the while it has been surmised that there is a synchronization problem anywhere in the CUDA code. This problem is the single biggest cause of the delay of the project.

5.5.2 Texture Memory

This is the memory type which is a remnant of the old precursor graphics cards. This is suitable for computational requirements meant for the display monitor. It may be used in general as well, but it is too slow to be useful for high performance computing. Each multiprocessor has 8 KB of memory. They are, however, cached and no access patterns and alignments are imposed. Coherence is also serious problem here.

5.5.3 Constant Memory

This is another memory that can be accessed by the host CPU. In fact only the CPU can write to this memory. Each multiprocessor has 64 KB. This is also cached. It is good if there are data that are not bound to be changed for the whole calculation.

5.5.4 Local Shared Memory

The local shared memory is the fastest to access without bank conflicts, as fast as the register access. This memory is arranged in 16 banks corresponding to the half-warp. The banks impose that each core accesses unique banks, or at most one conflict is tolerated only by accessing the same address. In fact, the latter rule can be generalized such that all of them just access only one data (or address). This is possible because the hardware supports broadcast of the first address conflict it sees. Then it delivers the conflict-free data. It does the broadcast-and-deliver routine as many times until it is able to service all accesses. The choice of which conflict to broadcast first from among the many cannot be determined. So it is important to avoid conflicts, but it can also be a powerful tool for broadcasting one data for all the processors in the block.

5.5.5 Registers

The registers are the fastest to access. They can be as fast as zero cycle, but they may be encountering conflicts in memory banks. Each multiprocessor has about 8192 registers.
shared by the different processors. Software variables are normally assigned to registers, and any excess is placed in local memory within and with comparable latencies as the global memory. Hence, the wise use of this memory is the key to efficiency. It is akin to an ace in the card game of blackjack.

Part III
OPERATING SYSTEM

Two operating systems on the HP machine are planned to be installed. One is the Red Hat Linux Enterprise version 4, hereafter referred to as OS-RH4. Another is the Ubuntu 8.04 Server Edition, also from now on called OS-U804. Both are for 64-bit machine. The early part of the experiment has experienced problems with OS-RH4. Eventually, only OS-U804 has been used. More about the problems will be discussed in the appropriate parts of the paper.

Part IV
SOFTWARE COMPILER AND TOOLS

The GPGPU Tesla is supported by a proprietary software architecture called CUDA or Computing Unified Device Architecture which is natively written for C. There is also a lower level language PTX, which is similar to an assembly language. The main compiler for CUDA and PTX into GPGPU Tesla executable code is NVCC provided by the same company.

The NVCC has a compilation mode which allows for the generation of CPU instructions to be run from the host machine instead of the GPGPU Tesla. This is called the emulation software. With this mode, we can insert printf’s to track down the flow of our programs. Note, however, that these printf’s must be enclosed in appropriate compiler directives to switch them off when the true compilation mode is set; otherwise, the compiler generates errors.

The company developed a debugger and profiler for Linux RedHat, but has not been tested by the author if it works with Ubuntu as well. Nevertheless, the profiler is not really the powerful one we observe with the code compiled for the CPU. The GPGPU Tesla does not support tracking the actual usertime in the different cores. It is also just the realtime measured by the clock function provided by the hardware. Any latency is not detectable by the function, more than what the ordinary realtime obtained by invoking the operating system provided time function can.

6 Description and Language

CUDA comes in two flavors, which correspond to the two levels of programming. The lower level is called CUDA utility. Basically, the higher level wraps several functionalities of the lower level and group them logically in a function so that some complexities of invoking the lower level are hidden. Generally the resulting code written in the lower level follow the same flow as the higher level. So if more control of the functionalities is needed, the lower level is more appropriate. In most instances, however, the higher level is sufficient. In fact in the implementation of the project, the whole set of CUDA is not implemented. Only the basic functionalities needed by the code are utilized.

PTX is more difficult to implement but it affords even more control. All CUDA codes reduce to PTX during compilation. In this manner, the PTX codes can be recompiled for different NVIDIA products.
7 Model and Architecture

The software architecture of CUDA closely resembles the hardware architecture. The basic unit of code for execution is called a kernel. In a session or running of the whole software, any number and kind of kernels may be invoked. Each invocation costs a nominal amount of time in milliseconds or so. From here, we will discuss the rest of the elements from the bottom up.

7.1 Thread

A thread is the smallest unit in the software architecture. Each thread is assigned to a core. Different threads, however, may be assigned to the same core. This obviously presents a problem with regards to the resources. To avoid conflicts, the hardware assigns separate resources for each thread, dividing the available resources available from the multiprocessor where the thread and the core belongs.

7.2 Block

A block is a group of thread organized in a logical fashion according to the computational requirements. A block can organize the threads in one, two, or three dimensions represented as x, y and z, to better provide support for the shape of the data. The maximum size of threads for each dimension is 512, 512, 64, respectively. Threads within a block can communicate through shared memory, and there are synchronization artifacts available for the block. Outside the block, the synchronization artifacts are not available.

Each thread in a block has a unique id corresponding to its logical order in the block. It also corresponds to the index of the actual core that supports it. The block is assigned to only one multiprocessor.

7.2.1 Warp

A warp is a group of threads in a block which the multiprocessor process at the same time in SIMD fashion. This actually reflects the hardware architecture where each execution unit of the multiprocessor takes four cycles. The maximum warp size is 32.

7.2.2 Half-Warp

The half-warp, which is half of the warp size, is the true correspondence to the hardware. It is a maximum of 16, which truly corresponds to the number of memory banks and size of the bus in the multiprocessor. In essence, the multiprocessor runs eight threads per cycle and 32 threads are run in one execution unit.

7.3 Grid

The grid is a group of same-sized blocks which can be handled in one kernel invocation. The blocks in a grid, however, do not have inter-communication and synchronization artifacts like the threads in the block. Like the blocks, the grid of blocks can have at most three dimensions, each has a maximum of 65535.

Each block in a grid has its own id. Threads as well have unique ids.
Figure 4: The figure shows one streaming multiprocessor with 8 processor cores and the maximum number of threads which can be spawned. These threads are the smallest units in the CUDA software architecture.
Part V

STATEMENT OF THE PROBLEM AND GOALS

How can the molecular simulation program be optimized using the GPGPU Tesla? The optimization shall be with respect to execution time, and optionally with respect to memory use and other concerns.

Part VI

SCOPE AND LIMITATION

Currently, the molecular simulation software is written in Fortran 77. The following shall be fully massively parallelized to run for NVIDIA Tesla C870.

1. The vector reciprocal square-root function. This function accepts a vector A. It outputs vector B whose entries are composed of the reciprocal square root of the corresponding entries in vector A. The corresponding vector square-root function is also provided.

2. The fastqin function. This function prepares the n-body calculation of distance between pairs of particles.

Due to the limitation of time, the inclusion of the following are covered in another paper. The main software used to run the molecular simulation shall be fully massively parallelized. Currently, the software is written in Fortran 77.

Part VII

SOFTWARE IMPLEMENTATION

8 Description

The work requires that the codes to be developed implement one for the parallelization of the rsqrt vector function, as well as the whole distance calculation of the Fastqin procedure. To comply with the two requirements, one software will be written for each, and correspondingly hereafter referred to as Phase 1 and Phase 2. Unless specified, the discussion to follow shall include software developed for both phases.

The software has been programmed to be aware as much as possible of the hardware that is available to it. It is able to detect automatically whether the GPGPU Tesla is present. It is also able to detect how many cores are available in the machine.

9 Software Models

The problem requires a better implementation of one isolated rsqrt function to be called by the program in Phase 1, and the likewise isolated procedure eventually calling the rsqrt function. The pieces of software developed are just libraries which will be linked to the original Fortran program.

9.1 Description

For each of the phases, there are three models of implementation applied corresponding to the available platform in the HP machine. They are as follows:
1. Serial model, which represents the normal software paradigm of implementing the algorithm to run in sequence;

2. Threaded model, which harnesses the multicore capacity of the HP machine, or any other machines such as those of the servers at UPPMAX; and

3. Cuda model, which in addition to the second model utilizes the power of GPGPU Tesla.

The final software for each phase is equipped with the three models. It can switch from one to pick the best way to execute the task at hand. For this purpose thresholds will be established and then utilized here for switching.

9.1.1 Threads

The last two models are actually tied up with each other. The threads are implemented in POSIX, and each thread is encapsulated by a software procedure. The procedure comes in two types: one for invoking a CPU core, and one for the GPGPU Tesla. The two shall be called CPU worker and CUDA worker, respectively. They are being managed by another software coordinator which is responsible for dividing the work for some or all of the workers. On initialization, after the software detects the presence of GPGPU Tesla and the CPU cores, it will create the corresponding worker to handle it. Note that the GPGPU Tesla is always assigned with one core to meet its host processing requirements, except when there are fewer CPU cores. Hence, the total number of workers will be the total number of GPGPU Tesla present or the CPU cores, whichever is bigger.

There is no reason to separate them because if all the CPU cores are utilized, the running of a task for each of them will mean that the processing for GPGPU Tesla will have to share with one of them. It defeats the purpose of employing the card. To make the captured core serviceable, the code for the GPGPU Tesla is made to run asynchronously with the CPU. So while some calculation is being done by the card, the core is also calculating the remaining task not yet done.

In the present implementation of the software, all the tasks are assigned to the other workers when the GPGPU Tesla is present. The other cores are supposed to proceed with the other calculations. However, the latter is beyond the scope of this thesis as it will touch upon the code it is not yet allowed to do.

9.1.2 Cuda Code

A separate library is written for Cuda code. One important aspect considered is that while data is delivered to the Cuda code as double precision, the GPGPU Tesla can process only single precision. There is no provision for conversion between data types. It has also been deemed unnecessary to write a specific code for the purpose considering there is already an available double precision GPGPU. It will also add complexity to the code so much because the GPGPU Tesla is not fully compliant with the current convention on conversion and representation; hence, two sets of code will have been written, one compliant to transfer data from the device to the host and non-compliant to interpret data from the host. Therefore, a portion of time is necessary for the double-to-single-to-double (DSD) conversion of data which is done serially in the host.

The data set does not normally fit in grid of blocks in multiple of 256, required to fill the GPGPU to full capacity calculation. And so chunking is employed. One big grid and one small grid, if necessary, are setup to invoke the kernel. The operations, however, are made asynchronous with the host CPU, so that while the GPGPU Tesla is running, the worker is also calculating the remaining data.
9.2 Experimental and Control Group of Softwares

To be able to evaluate the performance of the software developed, the original software is also utilized to obtain information that will be compared with the softwares developed. This will be referred to as the Fortran model. While it also runs serially, it will be distinguished from the Serial model because the latter is implemented in C. Compiler implementation is different for each language, and it is expected that this compiler differences will be reflected in the performance of each model. Furthermore, Phase 2 requires the rewriting of the whole Fastqin procedure. This will impact the comparison for the other models in that phase.

An experiment has been conducted to test the above comments, and it has been found out that indeed the Fortran model runs slightly faster than its counterpart Serial model. Noticeably, the usertime which is the indicator for the amount of machine instructions executed by the machine is lower for the Fortran model than for the Serial model. Hence in order to make a fair comparison, the Fortran model is utilized as the control for Phase 1, and the Serial model for Phase 2.

The Fortran model is preferred in Phase 1 because it is the natural gauge. There is only a slight difference. It is also not logical to implement another software serially only to find out that it is slower. Furthermore, the issue of the phase is to improve the performance of one isolated rsqrt function. No other part of the code is disturbed. So, the performance of any other software developed, will only be measured against that part of the code of the original program that does the same task.

The situation is different in Phase 2, however. Inside the Fastqin procedure are some codes preparing vectors or matrices of data and calculating distances. The implementation of this part of the code is compiler specific as discussed. Some compilers generate more machine instructions than others. To remove the compiler bias, the Serial model is used to gauge the other softwares developed. If the latter is written in Fortran as well, there is no doubt that the results will be comparable. This cannot be done easily, however, because the CUDA is natively implemented in C and the author is more experienced in the latter.

The experimental group consists of the different softwares and models developed.

10 Tests

10.1 Test Parameters

The test parameters utilized for comparisons are the realtime or the wall clock time, the usertime, and the accuracy of the calculations. The realtime is the time taken up by the running of the software model to its completion. However, this also includes the time performed by other tasks when the model is executing, and the time required to process some kernel or operating system calls. The latter is, however, negligible in most instances and so assumed to be zero in any calculation made in this paper. To minimize the interference of other programs, nothing else is allowed to be actively running while experiments are done. System tasks are normally unavoidable; but for others which can be turned off, they are allowed to run as it can be presumed that they are likewise negligible. The latter is also too cumbersome to turn on and off. However, as can be shown elsewhere below with the OS-RH4, they and the resulting hardware conflicts have affected so much the calculations that the early experimental results have been abandoned. The realtime of the control divided by the realtime of the experimental model is the speedup of the latter, and represents its performance. A value less than one means it is slower, and above one means it is faster than the control.

The usertime indicates how much machine instructions have been generated and executed. If the realtime is only accountable to the running of the model, as it is assumed here, then
the usertime divided by the realtime is an index of load balance achieved by the model, or efficiency of the latter in utilizing the full computational capability of the hardware.

Labels with suffix -r are normally runtime value, those with -u are usertime values.

10.2 Test Models

To perform preliminary tests, the Fortran code is replaced by a C code which simply accepts relevant parameters usually given by the Fortran code, then prepares the necessary test data according to the parameters provided, then invoke the software models for 10,000 times using the same parameters and data. The magnification is necessary as normally the time needed to perform the calculation is too small to be reflected in the realtime or usertime.

10.3 Simulation Test Case

Simulations have been run for molecules of $N = 100, 200, 300, 400, 800, 1200, 1600, \text{ and } 2000$. The simulation parameters are the ones provided by the supervisor of the thesis.

10.4 Statistical Tests

The main statistical parameter employed is the standard deviation from the mean. Depending on the result of the experiment, hypothesis tests will be utilized to decide the validity of the results. Also, analysis of variance, randomness tests, and other tests will be carried out if need be.

Part VIII
EXPERIMENTS

The experiments in general have been conducted in three parts. The first part is preliminary and exploratory, which serves to determine how the main part is going to proceed.

The second part - the basis part - is concerned with verifying the characteristics of the hardware being employed, the operating system chosen and the appropriate calculations obtained, as well as the interplay of the hardware and software components. This part also deals with the soundness of the software design. As the software involves the use of synchronization artifacts, tests on the presence of deadlocks are carried out. The algorithm is also tested on how much speed it can potentially deliver.

Finally, the third part defines the main substance of this thesis. Building on the previous experiments, the actual simulation experiments are finally undertaken. These are conducted in three stages for each of the software phases: the first is meant to determine the reliability of the runs; the second to find out the accuracy of the solution; and the third to run the simulations on the test case.

Note, however, that nothing here should be understood as to preclude the overlapping of any point of the experiments. They have not always been executed in sequence and problems normally define whether in the middle of a subdivision, the experiment has to go back to a particular instance indicated here as an early stage.

Note further that only the results of the last part are included and discussed in this paper. The detailed results of the first two parts will be the subject of a supplemental paper to be published in the future.
11 Molecular Simulation Phase 1

11.1 Reliability Test

Five dry run simulations have been conducted for each of the Cuda model and Fortran model. The respective means and standard deviation for each are then calculated. The results are presented in Table 1 on page 16.

11.2 Accuracy Test

Several results obtained from the Fortran model have been compared with several sets of results of the other simulation models. All of them have identical results. This is actually surprising as the computation in GPGPU Tesla is done in single precision, while the original sets of data are double precision. The Fortran model likewise operates in double precision.

11.3 Data Gathering

The actual simulations using the different models have been carried out using the test case. Figure 5 on page 17 summarizes the results.

12 Molecular Simulation Phase 2

12.1 Reliability Test

As in Phase 1, five dry run simulations have been conducted for each of the Cuda model and Fortran model. The respective means and standard deviation for each are also calculated. The results are presented in Table 1 on page 16.

12.2 Accuracy Test

Like in Phase 1, several results obtained from the Fortran model have been compared with several sets of results of the other simulation models. All of them have identical results. This may not apply absolutely to calculations made by GPGPU Tesla, because of the single-precision calculations of the coordinates.

12.3 Data Gathering

The actual simulations using the different models have been carried out using the test case. Figure 12 on page 28 summarizes the results.

Part IX

DISCUSSION AND ANALYSIS

13 Phase 1

13.1 Reliability and Accuracy Test

As tabulated the standard deviation from the mean of the realtime and usertime for each of the set of data corresponding to the Cuda model and the Fortran model are very nominal compared to the mean, even as the latter gets bigger. This indicates that the simulations and the test parameters obtained are very consistent, and reliable. It also means that no further statistical tests are necessary because all should capture the impressive consistency of the hardware, software, and even the time function used despite the fact that several system tasks have been running during the tests. The systimes outputted as well has proven that they can surely be neglected.
Table 1: Statistical Calculations for Phase 1. To test the reliability and accuracy of the simulation time of the experiments, two models have been run with varying number of beads 100, 200, 300 and 400, each run 5 times not consecutively. The average of 5 samples and the standard deviation are tabulated. The result shows that it is highly reliable and accurate even at 400 beads with less than 2 seconds deviation from an fairly high 21 minutes mean time. Hence, the cpu and the GPGPU Tesla card are highly consistent. This will surely apply in Phase 2 as well.

As noted in the Experiments, the solutions obtained for the experimental group are identical with the control.

13.2 Simulation Results

As shown in Figure [5] on page [17], the graphs are showing a superlinear pattern, at least. It may be cubic, as pointed out later. The figure shows three graphs for the time of three sets of simulations. The square-blue-line is for the serial Fortran model. It has the highest time to finish any simulation, except with 100 beads. The diamond-orange-line is for the threaded model. Initially, the model is following the square-blue-line in terms of the time to finish the simulation, except with 100 beads where it is the highest. With 2000 beads, however, the third graph becomes higher. The third graph is a ▽triangle-yellow-line for the CUDA model.
Figure 5: These are the different graphs for the time of three sets of simulations for three software models: one serial Fortran algorithm (square-blue-line), one threaded model parallelizing the rsqrt function (diamond-orange-line), and one CUDA model parallelizing the same function (▽triangle-yellow-line). Each set have varying number of beads shown on the x-axis. The time is shown on the y-axis.

The relationship of the results of the three model is further captured by the computation of the relative speedup from the serial Fortran model, plotted in Figure 6 on page 18. We see that the square-blue-line is a flat 1 ratio all the way, being the basis. The diamond-orange-line, starting at 1.96, rises fast and peaks at 1.6 with 400 beads. It then drops to 1.4 with 800 beads and diminishes very very slowly to 1.38 with 2000 beads. The ▽triangle-yellow-line begins at 1.22 and rises even faster peaking at 1.96 with 400 beads. It then drops to 1.81 with 800 beads and continues the trend not quite slowly to 1.33 with 2000 beads.
13.3 Benchmarking

The results of Phase 1 for the Cuda model are surprising. The GPGPU Tesla is supposed to run for a peak performance of 500 GFLOPS. The latter is a huge number comparable to servers at UPPMAX. So why the dismally low speedup? To answer the question, a special set of tests and calculations have been made. So benchmarking the GPGPU Tesla is in order.

13.4 The Challenge

Benchmarking and data analyses in Phase 1 allow us to reconstruct the timings of an rsqrt calculation, and decompose them into three components. Figure 7 on page 19 and Figure 8 on page 20 reflect the reconstruction. The first represents the situation where GPGPU Tesla is made to run asynchronously, allowing calculations also in the cpu; while the second for blocked execution. Clearly, the data load time from the host cpu to the device, and back, (shown as the bottom layer in blue) dwarfs the other two components (transfer of data from global memory to shared memory, and back, shown as the middle layer in orange, and calculations shown as the top layer in dark yellow) by about 5.7 orders of magnitude. As Phase 1 has shown, the composite time virtually determines the time of the simulation.

*The results of the benchmarking are presented in the supplemental report*
as \( N \) increases. Lowering this, correspondingly increases the speedup. Phase 2 addresses this issue.

Note that the inset shows the same date in percentage of the total time for a given number of beads. Disregarding any variations, about 85\% of the time is spent on communication cost between the host cpu and the GPGPU Tesla.

**Figure 7:** The reconstructed componentized timings with (asynchronous CUDA/cpu mode) of vector \( \text{rsqrt} \) operations, including double-to-float-and-back conversions. The varying sizes of the vector are given in the x-axis while the time is on the y-axis. The bottom layer in blue is the time to send a vector from the cpu to the global memory of the device and back. The second layer in orange is the corresponding time from the global memory to the local memory of the processor core. The top layer in dark yellow is the actual calculation time in the core. Note that the graph is not drawn to scale to show different scales on the x-axis. The inset presents the same data in percentage of the total time.
Figure 8: The reconstructed componentized timings with (blocked mode, when all calculations are done in the device) of vector rsqrt operations, including double-to-float-and-back conversions. The varying sizes of the vector are given in the x-axis while the time is on the y-axis. The bottom layer in blue is the time to send a vector from the CPU to the global memory of the device and back. The second layer in orange is the corresponding time from the global memory to the local memory of the processor core. The top layer in dark yellow is the actual calculation time in the core. Note that the graph is not drawn to scale to show different scales on the x-axis. The inset presents the same data in percentage of the total time.

13.5 Theoretical Considerations of the Results

In Phase 1, we have only obtained a speedup of about 1.81. We have expected more. Looking at the figures, however, we see that the bottleneck time, that is the time to transfer data from the CPU to the GPGPU Tesla card, has taken the lion share. So we examine how the data is created. The algorithm (as a C snippet) is as follows:

```
// ************************ MAIN PROGRAM ***************************
//*** comment: ny3 is fixed to 10
for (my3 = 0; my3 < ny3; my3++)
{
  //*** comment: ny2 is user given NPass not less than 100
  for (my2 = 0; my2 < ny2; my2++)
  {
    //*** comment: my1 is 10 x (number of beads - 1)
```

for (my1 = 0; my1 < ny1; my1++)
{
    //*** comment: il is the pivot, n1 is the number of beads
    il = kntot % (n1-1);
    kntot = kntot + 1;
    ...
    DoFastqin(n1,il);
    ...
}

DOFASTQIN PROCEDURE
******

void DoFastqin(int n1, int il)
{
...
int ind = 0;
for (i = 0; i < il; i++)
    for (j = il; j < n1; j++)
    {
        ind = ind + 1;
        xtemp = x6[i] - x6[j];
        ytemp = y6[i] - y6[j];
        ztemp = z6[i] - z6[j];
        r52[ind] = rkap2*(xtemp*xtemp + ytemp*ytemp + ztemp*ztemp);
    }
vrqrt(r52, eold, ind);
ind = 0;
for (i = 0; i < il; i++)
    for (j = il; j < n1; j++)
    {
        ind = ind + 1;
        xtemp = x6[i] - tx6[j];
        ytemp = y6[i] - ty6[j];
        ztemp = z6[i] - tz6[j];
        r52[ind] = rkap2*(xtemp*xtemp + ytemp*ytemp + ztemp*ztemp);
    }
vrqrt(r52, enew, ind);
for (i2 = 0; i2 < ind; i2++)
    delta = delta + enew[i] - eold[i];
...
}

Lines 14 and 15 of the algorithm controls the pivot. Whatever bead it starts with, it just
basically loops around sequentially within the range bead number 1 and the penultimate
N-1. Line 9 tells us that it precisely involves a factor of N-1 iterations. Hence, we will
treat the algorithm as a set of pivots from 1 to N-1. Note that the outermost loop in Line
3 involves 10 iterations, and the innermost loop in Line 9 has 10 sets of pivots, or 100 sets.
However, it cancels with the 100 as a divisor of NPass in the middle loop in Line 7. As
NPass=4000 in the simulation, the whole set is then repeated 4000 times.

Note further that Lines 38 and 49 are the actual extraction of the reciprocal square roots.
All the others between Line 25 and Line 49 are the preparation for distance calculations.
In a simulation of $N$ molecules or beads, we assign a pivot bead from 1 to $N-1$. While fixing the beads up to the pivot bead, we let the other beads after the pivot to take new positions. We then determine whether the energy of the new configuration of the molecules is lower than the old configuration. To do this, we need to calculate for each of the configurations the pair distances of each bead before and including the pivot bead with the rest of the beads.

So, the number of pair distances to be computed is equal to the product of the number of beads up to the pivot and those after. We set up the distance calculations which require one rsqrt operation per pair distance. The rsqrt operations are then delegated to the GPGPU Tesla card.

To determine the data loaded, we have the following equations. Let $L$ be the number of beads up to the pivot. Let say also that the total rsqrt operations needed is represented by the variable $R_1$. Then we have the following derivations to get $R_1$.

For a single configuration

$$r = \sum_{L=1}^{N-1} L (N - L)$$

$$= \sum_{L=1}^{N-1} (LN - L^2)$$

$$= \sum_{L=1}^{N-1} LN - \sum_{L=1}^{N-1} L^2$$

$$= N \sum_{L=1}^{N-1} L - \sum_{L=1}^{N-1} L^2$$

$$= N (N - 1) \frac{N}{2} - (N - 1) N \left( \frac{2[N-1] + 1}{6} \right)$$

$$= \left( \frac{(N - 1) N}{2} \right) \left( N - \frac{2N-1}{3} \right)$$

$$= \left( \frac{(N - 1) N}{2} \right) \left( \frac{(N+1)}{3} \right)$$

$$= \frac{(N^2 - 1) N}{6}$$

But we need two configurations, therefore

$$R_1 = 2r = \frac{(N^2 - 1) N}{3} \quad (2)$$

In phase 2, on the other hand, the whole pair distance calculations are assigned to the GPGPU Tesla card. So we load the coordinates required. There are three coordinates per bead. Note that we have $(N - L)$ coordinates for the new configuration. The total number of coordinates or data $R_2$
For a single coordinate

\[
\begin{align*}
 r &= \sum_{L=1}^{N-1} (N + (N - L)) \\
 &= \sum_{L=1}^{N-1} 2N - \sum_{L=1}^{N-1} L \\
 &= 2N(N - 1) - \left(\frac{(N - 1)N}{2}\right) \\
 &= N(N - 1) \left(2 - \frac{1}{2}\right) \\
 &= \frac{3(N - 1)N}{2}
\end{align*}
\]

For three coordinates, therefore

\[
R_2 = 3r = \frac{9(N - 1)N}{2}
\]

We know that the increase in time needed to load is linear with respect to the amount of data loaded, except for small amounts. Assuming linearity, the ratio of the \(R_1\) over \(R_2\) indicates the speedup \(S\) on this part of the code, which is most dominating.

\[
S \propto \frac{R_1}{R_2} \propto \frac{(N^2 - 1)N}{9(N - 1)N^2}
\]

Hence,

\[
S \propto \frac{2(N + 1)}{27} \quad (3)
\]

The proportionality is removed only by multiplying a time factor related to the GPGPU Tesla.

Figure 9 on page 24 is a graph of the of the theoretical data load time in Phase 2 for the corresponding value in Phase 1. As the speedup is linear with \(N + 1\), the decrease is also inversely proportional to the same parameter.
Figure 9: The ideal data load round trip time for the varying vectors in Phase II. The data from Phase I has been divided by the theoretical speedup derived in \( \text{previous page} \) on the previous page. Note the graph is not drawn to scale to show different scales on the x-axis.

Figure 10 on page 25 and Figure 11 on page 26 depict the change in time for each of the three components that make up the runtime of the GPGPU Tesla, after the data load time has been adjusted. Note that by sending only the coordinates and doing all the preparation for the distance calculation in the GPGPU Tesla as well, reduces the communication cost dramatically. In fact, compared to the top yellow layer and the middle orange layer, the bottom blue layer is trivialized. By looking at the insets, we see that the bottom blue layer is visible only with the smallest data size (vector) starting at about 20% and diminishing to near zero with about a data size of 2000.

As will be noted below, these pictures are still bound to change with the introduction of other factors that will affect the time of the new dominant components, especially the algorithm to be employed.
Figure 10: The adjusted (applying results in Figure 9 on page 24) componentized timings with (asynchronous CUDA/cpu mode) of vector rsqrt operations, including double-to-float-and-back conversions. The varying sizes of the vector are given in the x-axis while the time is on the y-axis. The bottom layer in blue is the time to send a vector from the the cpu to the global memory of the device and back. The second layer in orange is the corresponding time from the global memory to the local memory of the processor core. The top layer in dark yellow is the actual calculation time in the core. Note that the graph is not drawn to scale to show different scales on the x-axis. The inset presents the same data in percentage of the total time.
Figure 11: The adjusted (applying results in Figure 9 on page 24) componentized timings with (blocked mode, when all calculations are done in the device) of vector rsqrt operations, including double-to-float-and-back conversions. The varying sizes of the vector are given in the x-axis while the time is on the y-axis. The bottom layer in blue is the time to send a vector from the CPU to the global memory of the device and back. The second layer in orange is the corresponding time from the global memory to the local memory of the processor core. The top layer in dark yellow is the actual calculation time in the core. Note that the graph is not drawn to scale to show different scales on the x-axis. The inset presents the same data in percentage of the total time.

Note that for phase 2, there is a small increase in time for the setting up of distance calculations before the rsqrt are taken. There is, however, a decrease due to local block reductions and global reduction which are done in parallel. Moreover, the global to local memory transfer in the card will be affected also by the blocking introduced in the algorithm. Without blocking, the time for the transfer will increase by more than three-folds because the total transfer will be equal to $N$ multiplied by three coordinates, plus the addition of three coordinates per bead in the new configuration. As blocking is increased as it is in the algorithm employed, the speedup for this part will correspondingly decrease linearly or so. The speedup is then expected to be more.

In phase 1, on the other hand, the GPGPU Tesla is expected to output the whole vector of rsqrt values. This reasonably doubles the speedup because phase 2 is required to output only 1 value which is the result of the reduction. To reflect this to the S is not fair because the requirement is attributable neither to the property of the card nor to the goodness of the algorithm. However, from the point of view of the result and graphical interpretations, the widening effect of this requirement is very visible.
14 Phase 2

14.1 Reliability and Accuracy Test

Table 2 on page 27 contains the data and calculations used to determine the reliability of
the softwares used for the simulation. Like in Phase 1, the standard deviations suggest no
need for further statistical tests. The simulation will be reliable.

Also noted in the Experiments, the solutions obtained for the experimental group (Cuda
model not absolutely compliant) are identical with the control.

<table>
<thead>
<tr>
<th>SET</th>
<th>N</th>
<th>Thread-r</th>
<th>Thread-u</th>
<th>Fortran-r</th>
<th>Fortran-u</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>100</td>
<td>00:02:22.39</td>
<td>00:00:32.41</td>
<td>00:00:30.38</td>
<td>00:00:30.36</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>00:02:23.75</td>
<td>00:04:21.52</td>
<td>00:04:09.94</td>
<td>00:04:09.69</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>00:07:29.43</td>
<td>00:14:11.88</td>
<td>00:13:25.49</td>
<td>00:13:24.68</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>100</td>
<td>00:00:22.18</td>
<td>00:00:32.86</td>
<td>00:00:30.35</td>
<td>00:00:30.26</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>00:02:23.30</td>
<td>00:04:21.59</td>
<td>00:03:59.62</td>
<td>00:03:59.39</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td>00:07:29.68</td>
<td>00:14:10.96</td>
<td>00:13:53.75</td>
<td>00:13:53.15</td>
</tr>
<tr>
<td></td>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>100</td>
<td>00:00:22.39</td>
<td>00:00:32.86</td>
<td>00:00:30.35</td>
<td>00:00:30.27</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>00:02:23.68</td>
<td>00:04:22.46</td>
<td>00:04:13.05</td>
<td>00:04:12.82</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>100</td>
<td>00:00:22.18</td>
<td>00:00:33.11</td>
<td>00:00:30.36</td>
<td>00:00:30.31</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>00:02:23.71</td>
<td>00:04:21.22</td>
<td>00:04:09.37</td>
<td>00:04:09.07</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>100</td>
<td>00:00:22.31</td>
<td>00:00:32.99</td>
<td>00:00:33.87</td>
<td>00:00:33.86</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>00:02:24.08</td>
<td>00:04:21.74</td>
<td>00:04:11.75</td>
<td>00:04:11.46</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STDDEV</td>
<td>100</td>
<td>00:00:00.10</td>
<td>00:00:00.27</td>
<td>00:00:01.57</td>
<td>00:00:01.59</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>00:00:00.21</td>
<td>00:00:00.46</td>
<td>00:00:05.31</td>
<td>00:00:05.30</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MEAN</td>
<td>100</td>
<td>00:00:22.29</td>
<td>00:00:32.85</td>
<td>00:00:31.06</td>
<td>00:00:31.01</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>00:02:23.74</td>
<td>00:04:21.71</td>
<td>00:04:08.75</td>
<td>00:04:08.49</td>
</tr>
<tr>
<td></td>
<td>300</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>400</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 2: Statistical Calculations for Phase 2. To test the reliability and accuracy of the simulation time
of the experiments, two models have been run with varying number of beads 100, 200, 300 and 400, each
run 5 times not consecutively. The average of 5 samples and the standard deviation are tabulated. The
result shows that it is highly reliable and accurate as expected. Some results have been missing, but as
already established in phase 1, the cpu and the GPGPU Tesla card are highly consistent.

14.2 Simulation Results

Like in Phase 1, the graphs in Figure 12 on page 28 are depicting a superlinear pattern.
Unlike Phase 1, however, there is now a clear separation of the three graphs. Still, the
serial C model has the highest timings, as shown by the square-blue-line. The threaded
model in diamond-orange-line follows at about half the time. The \(\triangleleft\)triangle-yellow-line of the CUDA model almost lies flat on the horizontal axis.

Figure 12: These are the different graphs for the time of three sets of simulations for three software models: one serial C algorithm (square-blue-line), one threaded model parallelizing the fastqin function (diamond-orange-line), and one CUDA model parallelizing the same function (\(\triangleleft\)triangle-yellow-line). Each set have varying number of beads shown on the x-axis. The time is shown on the y-axis.

The relationship of the results can be examined further by computing the relative speedup from the serial C model, plotted in Figure 13 on page 29. Again, the serial model is made the basis, so that the square-blue-line is still 1 parallel to the horizontal axis. The diamond-orange-line rises to about 1.92, more or less, remains to that ratio starting with 800 beads. But the \(\triangleleft\)triangle-yellow-line, though starting only from 400 beads at 1.33, continues to climb almost linearly until it reaches 13.13 with 2000 beads. Note that this is consistent with the theoretical speedup of CUDA as established in 3 on page 23.
Figure 13: These are the graphs of the speedup of the Phase 1 results presented in Figure 12 on page 28. The result for the serial C model has been used as the basis or numerator for the speedup ratio. The varying number of beads are shown on the x-axis while the ratio is on the y-axis.

Part X

CONCLUSION

Figure 14 on page 30 is the plot of all the simulation results of both phases of the experiment. As we can clearly see, the graphs of the serial models in phase 1 and phase 2 which are the square-blue-line and △triangle-green-line, respectively, have the highest timings serving as the upper bound of the other graphs. It is also evident that the CUDA phase 2 model in ◁triangle-cyan-line for the parallelized fastqin gives the best results. The other three: the threaded models for phase 1 and phase 2 in diamond-orange-line and ▶triangle-maroon-line, and the CUDA phase 1 model in ▼triangle-yellow-line lie close to each other.
Figure 14: The combined simulation results of two phases in Figure 5 on page 17 and Figure 12 on page 28. Note the motifs for the graphs of phase 2 have been changed to △triangle-green for the fqSerial model, ⊿triangle-maroon for fqThreaded, and ◀triangle-cyan for the fqCUDA model. For phase 1, it is the same as in the previous figure. Still, the number of beads are on the x-axis and the time on the y-axis.

Figure 15 on page 32 shows us the relative speedup of the different models. For the threaded models, when only the rsqrt function is parallelized, the speedup settles to 1.38. When the whole fastqin function is parallelized, the ratio of 1.92 has been achieved. Clearly, the preparation for the distance calculations is itself a significant portion of the algorithm. It certainly drives down the performance if not parallelized. How much this is in the algorithm in terms of total time can be calculated using the Amdahl’s Law. The latter can be expressed as follows:

\[
R = \frac{1}{S + \frac{P}{n}}
\]

where \(R\) is the speedup ratio, \(S\) is the fraction of time accounted to the serial part of the code, \(P\) is the corresponding fraction of time attributed to the parallelized portion, using \(n\) processors. \(S + P = 1\)

If we divide the serial part into that of the preparation for the distance calculation, \(S_d\), and \(S_0\) for the rest, then we have:
\[ R = \frac{1}{S_0 + S_d + \frac{P}{n}} \]

or using the data of threaded Phase 1 model:

\[ 1.38 = \frac{1}{S_0 + S_d + \frac{P}{2}} \]  \hspace{1cm} (4)

For Phase 2, we also parallelized \( S_d \). Hence, the speedup will now be expressed as:

\[ R = \frac{1}{S_0 + \frac{P + S_d}{n}} \]

or using the data of threaded Phase 2 model:

\[ 1.92 = \frac{1}{S_0 + \frac{P + S_d}{2}} \]  \hspace{1cm} (5)

Solving for \( S_d \), we combine 4 and 5 to get

\[ S_0 + S_d + \frac{P}{2} = \frac{1}{1.38} \]

\[ S_0 + S_d + \frac{P}{2} = \frac{1}{1.92} \]

Then we get,

\[ S_d = 0.41 \]

\[ P = 0.55 \]

\[ S_0 = 0.04 \]

Hence, the preparation for the distance calculations takes about 41\% of the time.
Figure 15: The combined simulation results of two phases in Figure 6 on page 18 and Figure 13 on page 29. Note the motifs for the graphs of phase 2 have been changed to △triangle-green for the fqThreaded model, and ◁triangle-cyan for fqCUDA. fqCUDA is also compared with the Fortran model in phase 1 and the resulting graph of the ratio is presented as ▶triangle-maroon for the fqCUDA model/P1. For phase 1, it is the same as in the previous figure. Still, the number of beads are on the x-axis and the ratio on the y-axis.

The CUDA models have very surprising results. The graph for the Phase 1 model rises very early and drops continuously. What are the factors for this behavior? There are at least two main reasons. First is the 41% preparation time for distance calculations. Second, and more importantly, is the bandwidth issue. The latter is about 85% of the whole calculation time in CUDA.

The graphs for the CUDA Phase 2 model have decisively shown that addressing the two issues have resulted in an increasing speedup almost linearly. With 2000 beads, the speedup is 13.13. Beyond 2000 beads, the trend will still continue until 8000 beads. With these pieces of information, it will take about 21 hours to complete the simulation with 4000 beads, and 3½ days with 8000 beads. Equivalently, a speedup of about 49.37 with 4000 beads and 191.83 with 8000 beads will be achieved. With these figures it will have taken the serial model 43 days to finish the simulation with 4000 beads and 96 weeks (about 2 years) with 8000 beads. Note, however, that the numbers become more and more speculative as the number of beads increases.

Therefore, the original problem which was thought to be virtually dominated by the vast amount of time spent for the rsqrt function has been converted: first, as a problem simply of
the parallelization of the preparation for the distance calculations; and second, as a problem of managing the bandwidth problem. That is exactly how the molecular simulation has been optimized using the GPGPU Tesla, which is the goal of this paper. The software models deliver asymptotic linear speedup and proper load balance.

It may be asked, will it be possible to increase the CPU's and CPU cores to address the same issues? Of course, we can always do that! But as the Figure [15] on page 32 conveys, it will need a lot of CPU's to equal the performance of just one GPGPU Tesla. Economics-wise, one GPGPU Tesla priced at about the same as one computer workstation will definitely be more preferable. Add the other factors such as power consumption cost, and space requirement (a scientist can put the GPGPU powered computer in his or her table!), and maintenance, among others, all on the side of GPGPU Tesla.

Clearly, a well-written software model and GPGPU Tesla make up a team of perfect scalability.

Part XI

RECOMMENDATION FOR FUTURE WORK

Figure [16] on page 34 is a projection of the time needed to complete the molecular simulations involving the number of beads beyond those actually dealt with in the experiments. The square-blue-line in the figure is the time (in seconds) distribution of the mid-pivots with the respective number of beads. The diamond-orange-line is the graph of the corresponding projected simulation time in hours. Sufficient tests up to 4000 beads can be conducted using existing software models. Increasingly scant information may be obtained up to 8000 beads. Extrapolation which is guided by earlier results and established theoretical considerations can only be employed to reconstruct the time it will take to conduct a simulation up to 16000 beads. Admittedly, however, beyond the full capability of the GPGPU the simulation time will go back to its cubic character. The speedup in this part should be about constant, ideally the linear speedup equivalent to the active number of cores.
Figure 16: The graphs are the projected simulation time for the varying number of beads. The square-blue line represents calculation time in seconds for the mid-pivot or half the number of beads. The time is based on experimental results up to 4000. The rest is extrapolated linearly (caution: there are other factors relating to the GPGPU Tesla and cpu that may affect this).

The diamond-orange line is the projected simulation time for the particular number of beads. Projection is based on the 1:1 algorithm ratio which ideally produces a runtime profile of an isosceles triangle peaking at the midpoint. Experiments, however, have shown that this assumption is only asymptotically true. Other factors relating to the GPGPU Tesla have come into the picture to distort the shape with runtime deviation up to around 10% more.

The graphs are the reflection of an ideal scenario. This means that for individual pivots in a simulation with a given number of beads, the mid-pivot takes up the highest time, or the peak-time. The peak-times follows a linear distribution vis-a-vis the number of beads. It is also presumed that from the mid-pivot, the distribution of time to finish calculations for the pivots on either side of the mid-pivot is linear approaching zero. Hence, the total graph formed, if adjacent datapoints are connected, is that of an isosceles triangle sitting on the horizontal axis.

With proper implementation of the algorithm, the ideal numbers are achievable. Aside from the well-written algorithm, there are now two new generations of NVIDIA cards. These are the NVIDIA Tesla T10 series, and the NVIDIA Fermi series (to be released anytime now). They will be summarized in Table 3 on page 36 below. Their better specifications and their double-precision capabilities are certainly going to drive the speedup even higher to meet the ideal scenario much more easily.
If serviceability of the algorithm cannot be utilized to increase the number of beads, it is still possible to increase it by designing algorithms to iteratively chunk the required task and perform the calculations. How will this impact the time? Figure 17 on page 35 provides the actual number of distance and rsqrt calculations with varying number of beads. It is cubic based on the formula established in 2 on page 22. Note, however, that each point in the graph is not delivered in one task. They are delivered in small batches. Ideally, each batch is apportioned among the available GPGPU’s.

Figure 17: The graph shows the total number of rsqrt and distance calculations done within the fastqin function for a 4000-repetition many-body simulation. The varying number of beads are on the x-axis and the number of calculations on the y-axis. This is actually a cubic graph.

Whether or not the graphs can be extended as in the ideal scenario with just one GPGPU card, there is likewise another way to increase the number of beads in the simulation. This way is to increase the number of GPGPU’s. This can be done by providing additional separate GPGPU cards, or by putting one or more of the server editions which are actually a bunch of 4 GPGPU cards in one connection to the host machine. Ideally, there should be also a linear speedup with respect to the number of GPGPU’s added; which means that if it takes T units of time to perform a simulation with N beads, then it should take only T/g units of time to do it with g GPGPU’s. But again, this can be a tricky business, especially with the bandwidth issue.

If optimum performance is necessary, the the whole software should be parallelized. While some parts of the code are absolutely serial, it is just alright as the difference in clock or
speed of the host machine and GPGPU card is not significant. There is just one point to consider when doing the parallelization. The generation of random numbers as specified in this simulation could not easily be parallelized. Literature abounds, however, with techniques and methods to provide random numbers in a parallel environment.

Part XII
SUMMARY OF NVIDIA GPGPU Cards

<table>
<thead>
<tr>
<th>NVIDIA GPGPU</th>
<th>C870</th>
<th>T10</th>
<th>Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Streaming Multiprocessor (SM)</td>
<td>16</td>
<td>30</td>
<td>16</td>
</tr>
<tr>
<td>Cores Per SM</td>
<td>8</td>
<td>8</td>
<td>32</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>Double Precision Floating Point</td>
<td>None</td>
<td>30 MAD ops/clock</td>
<td>256 FMA ops/clock</td>
</tr>
<tr>
<td>Single Precision Floating Point</td>
<td>128 MAD ops/clock</td>
<td>240 MAD ops/clock</td>
<td>512 FMA ops/clock</td>
</tr>
<tr>
<td>Special Function Units (SFU)/SM</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Warp schedulers (per SM)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Shared Memory (per SM)</td>
<td>16 KB</td>
<td>16 KB</td>
<td>48 KB or 16 KB</td>
</tr>
<tr>
<td>L1 Cache (per SM)</td>
<td>None</td>
<td>None</td>
<td>16 KB or 48 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>None</td>
<td>None</td>
<td>768 KB</td>
</tr>
<tr>
<td>Concurrent Kernels (multitasking)</td>
<td>No</td>
<td>No</td>
<td>Up to 16</td>
</tr>
<tr>
<td>ECC Memory Support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Load/Store Address Width</td>
<td>32-bit</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
</tbody>
</table>

Table 3: This is the tabulation of some of the most important specifications of GPGPU’s available (or soon to be) from NVIDIA. The paper uses the C870. Note that only those for high performance computing cards are included. There are also CUDA enabled graphics cards which, in addition to their graphics capabilities, can be programmed just like the any other GPGPU’s. MAD means multiply-and-add operation. FMA means Fermi-Multiply-and-Add operation. Both is done in one clock but only FMA has full precision. Note that Fermi is still going to be released soon. Note also that the double precision capability of Fermi, alone, is already twice the single precision of GPGPU Tesla utilized in this paper.

References

[PTX] NVIDIA Compute, PTX: Parallel Thread Execution.
[Developer Tech] Optimizing CUDA, by Mark Harris, NVIDIA Developer Technology.

Fermi Whitepaper, p. 11. Please see the documentation for more details.