Department of Computing

Imperial College London

Optimised Kinetic Simulation Of Muscles

Author:
Alexander Karapetian
ak6309@doc.ic.ac.uk

Medical Supervisors:
Prof. Nancy Curtin
n.curtin@imperial.ac.uk

Prof. Roger Woledge
r.woledge@imperial.ac.uk

Supervisor:
Prof. Wayne Luk
wl@doc.ic.ac.uk

Second Marker:
Prof. Duncan Gillies
d.gillies@imperial.ac.uk

Submitted in partial fulfilment of the requirements for the BEng Degree in Computing of Imperial College London

June 19, 2012

Abstract

Muscle contraction is an important part of locomotion and many other essential functions. The Sliding Filament Model describes active shortening of muscles as the result of myosin crossbridges causing the myosin filaments to slide relative to actin filaments. In this project, we optimise and accelerate a computer simulation of a neighbourhood of crossbridges which are treated as a dependent population. We subsequently obtain novel results describing the effects of more compliant environments on the life of a population of crossbridges. Since the filaments stretch when a force is applied, a movement appears to be produced across them. Using existing ideas about how each crossbridge reacts to relative movement of its two ends and current measurements of the compliance of the filaments and attached crossbridges, we show that the dynamics of a population of crossbridges along the myosin filament are affected by interactions with its neighbours and present evidence to suggest that a higher compliance leads to a shorter crossbridge lifespan. The stochastic model of this behaviour is time consuming when executed on a large scale. We present an optimised version of the simulation in a compiled language and demonstrate performance gains of approximately 3000 times the original run time using acceleration and parallelism. From many long term runs of this simulation with our performance gains, we are able to present never before seen results to benefit our understanding of muscle contraction.

This work is dedicated to Sir Andrew F. Huxley
22 November 1917 – 30 May 2012

Acknowledgements

I would first like to thank Prof. Wayne Luk for his help and support during this project as supervisor. His extended contributions and decision to send me on a Maxeler training day allowed me to better understand the high performance computing tools I was working with. I would also like to thank Prof. Duncan Gillies as the second marker for my project for his patience and help in evaluating my progress at its early stages.

I would like to thank my medical supervisors, Prof. Roger Woledge and Prof. Nancy Curtin, to whom I am greatly indebted for their constant encouragement, guidance and direction. Their quick feedback and explanations of the biological theory which underpins the simulations involved were crucial in aiding my understanding of their work. I would also like to thank Jonathan Evans, also from Imperial College London, for discussing ideas with me, providing reading material and suggestions to improve my approaches to various problems in the project.

Finally, I would like to extend my gratitude to my friends and family for their unconditional support and encouragement and would like to thank those not mentioned that provided their assistance with suggestions and direction for this project.

Contents

1 Introduction **5**
- 1.1 Problem Motivation . . . 6
- 1.2 Problem Specification . . . 7
- 1.3 Project Objectives & Achievements . . . 8
- 1.4 Summary . . . 9

2 Background **11**
- 2.1 Sliding Filament Model . . . 11
- 2.2 Muscle Contraction . . . 13
- 2.3 Force–Velocity Relationship . . . 15
- 2.4 Probabilistic Models . . . 16
  - 2.4.1 Complexity Theory . . . 16
- 2.5 Overview of Acceleration Methods . . . 18
  - 2.5.1 Acceleration . . . 19
  - 2.5.2 Reconfigurable Hardware . . . 19
  - 2.5.3 General Purpose Graphics Processing Units . . . 28
  - 2.5.4 High Throughput Computing . . . 28
- 2.6 Summary . . . 30

3 Optimising the KMC Simulation in MATLAB **32**
- 3.1 The KMC Simulation . . . 32
- 3.2 Analysing the MATLAB script . . . 35
- 3.3 Programming Language Choices . . . 36
- 3.4 Simulink Coder . . . 37
  - 3.4.1 Correctness . . . 38
  - 3.4.2 Custom Types . . . 39
  - 3.4.3 Pseudo-Random Number Generator . . . 40
- 3.5 Optimisation Gains . . . 41
- 3.6 Summary . . . 41

4 Optimising the KMC Simulation in C++ **43**
- 4.1 Refactoring . . . 44
- 4.2 Profiling . . . 45
- 4.3 Reimplementing MATLAB Functions . . . 46

4.4 Functionality & Performance Gains 48
4.5 Experimental Considerations 49
4.6 Summary 50

5 Accelerating the KMC Simulation **51**
5.1 Resource Considerations 51
5.2 FPGA Acceleration 52
5.3 High Throughput Acceleration 53
5.3.1 Correctness 54
5.4 Profiling 56
5.5 Summary 58

6 Experimental Analysis **60**
6.1 Voluminous File Handling 60
6.2 Interpolation 62
6.3 Experimental Results 63
6.4 Summary 65

7 Conclusions **68**
7.1 Project Review 68
7.2 Future Work 70
7.2.1 Medical Research Aspects 70
7.2.2 Computing Research Aspects 70
7.3 Closing Remarks 71

Chapter 1

Introduction

Understanding the process by which muscles contract has been a challenge for researchers in Medicine since the introduction of the Sliding Filament Theory by A. F. Huxley[1] in 1954. Complex biological systems such as muscles typically contain significantly large amounts of microscopic structures which work together to produce the activity we know as contraction. It is possible to simulate such activity using mathematical methods to learn more about the underlying processes and how they behave over long periods of time. Mathematically modelling such a complex system, however, requires us to delve into its fundamental constituent parts, and the vast number of variables present results in large amounts of heavy computation to be performed. The consequence is that the simulations are restricted in how accurately they represent the underlying system and are not easily scalable to run for longer periods of time.

Computer hardware has improved considerably over time[2]. We have seen vast performance gains when running programs over previous generations. This is particularly due to the advent of multi-core systems, but their potential for parallelism must be leveraged by the software developer to take full advantage. In addition, supercomputing hardware, though still niche, has begun to increase in availability[3]. These advancements provide crucial additions to the inventory available to developers for speeding up algorithms. Our simulation is one such algorithm.

In this project, we look at various techniques to optimise, accelerate and further develop a computer simulation for muscle contraction as described by the Sliding Filament Theory. We explore the correctness of the computer model and its scalability to a large number of runs while maintaining accuracy. We use these runs to present previously unmeasured evidence suggesting that the lifespan of a population of crossbridges are affected by the compliance of their surrounding filaments.

This report contributes the following:

- An optimised version of the simulation in MATLAB (chapter 3).
- An optimised version of the simulation in C++ (chapter 4).

- An accelerated version of the simulation 2976 times faster than the original with greater accuracy (chapter 5).
- Analysis of the results which provides novel evidence of the dependence of crossbridges on compliance (chapter 6).

1.1 Problem Motivation

Simulations of muscle activity are often simplified by treating a muscle as a material with certain physical tensile stress and strain values, though this characterisation of muscles as polymers is a macroscopic and unrealistic view of the actual system in place. In this project, we consider a simulation that considers the system of muscle cells at the microscopic level. Professors Roger Woledge and Nancy Curtin, from the National Heart and Lung Institute (Imperial College London), have leveraged previous work to develop a method of simulating the activity between filaments correctly without resorting to the standard method of numerical integration of the differential equations involved or by simplifying the muscle's actions.

The Sliding Filament Theory describes two filaments which contribute to muscle contraction, the thick (myosin) and thin (actin) filaments. The thick filament consists of several myosin heads which attach to the thin filament, and by doing so form crossbridges (XBs). This action of XBs binding causes the thin filament to slide, and in turn is the key component responsible for muscle contraction.

It is generally been assumed in other simulations that the XBs act independently without influence from neighbouring XBs. However, we know that the filaments are compliant and will stretch when a force is applied. Because of this, we know a movement will be produced across the filament. We have good measurements of the compliance of the filaments and attached XBs, and we also have well formed ideas about how each XB reacts to relative movement of its two ends. The simulation by the medical supervisors to this project, R. C. Woledge and N. A. Curtin, describes a population of XBs and provides forces at each time interval up to a certain point as output.

The results from the simulation can be used to make observations about the XBs interacting between the two filaments in order to allow contraction. Since the simulation is computationally complex, this behaviour has never been modelled successfully. We explore the extent to which a XB can act independently in this project by optimising and accelerating the simulation to be able to obtain sufficient results. While most simulations at this level generally assume each XB to be independent, our method looks at the filaments' movements and compliance when stretched. The movement caused by filaments during stretching appears to affect the nearby XBs. Using this information, we learn that the dynamics of a population of XBs is affected by interactions with neighbouring XBs.

Our long term runs of the optimised simulation produces novel results which

demonstrate that a higher compliance in the actin and myosin filaments have an effect on the XBs ability to carry force. We present evidence in chapter 6 which, for the first time, experimentally suggests that environments with a higher compliance result in the XBs having a lower lifespan.

To perform this analysis, we first look at the simulation and examine its long term behaviour. This involves running the program a significantly large number of times. We use the initial target of 1,000,000 to ensure long term behaviour has been established. We then use the output from a million runs of the simulation to check the extent to which an XB can act independently, an aspect we believe has not been tested prior to this work.

To be able to achieve our target of a million runs, we first optimise the simulation to a state which takes full advantage of the hardware it is running on. We use various methods to do this, including refactoring, parallelism and optimisation, outlined in chapters 3 and 4. We explore the benefits from accelerating using Field Programmable Gate Arrays (FPGAs) and High Throughput Computing systems and present our findings as points for discussion in chapters 5 and 6.

1.2 Problem Specification

To be able to produce previously unmeasured results, we must run the simulation a large amount of times. We target an amount of 1 million runs in this project. We know that the time taken to run the simulation once is approximately 5s on a Intel Core 2 Duo laptop clocked at 2.27GHz with 4GB RAM running 64-bit MATLAB R2011b. This value varies based on the computer's hardware specifications. A maximum run time of 11s was found after testing across various modern PC and laptop hardware.

The worst case scenario for execution time of 11s is extrapolated in Figure 1.1 to show that in its original form, the KMC simulation would take 1 day to complete 10,000 runs, 1 week for 100,000 and 4 months for the target 1,000,000 runs. We therefore confirm that the original simulation is not scalable and therefore needs to be optimised and/or accelerated.

In this project we show that there are optimisations that can be performed within MATLAB to speed up the simulation's execution time (chapter 3). We also demonstrate that migrating from an interpreted to a compiled language such as C++ provides extensive performance gains (chapter 4). We also explore various acceleration methods to speed up the simulation to be able to run 1,000,000 runs in a reasonable time, providing us with significantly greater performance than the original (chapter 5). We target a maximum execution time of one day and show that this is not only attainable with a combination of optimisation and acceleration, but that a maximum execution time under three hours is attained using our methods, making our methods 2976 times faster.

To be able to reach this target we must first identify the various bottlenecks present in the existing code, and explore languages that can serve as platforms to improve the original simulation upon.

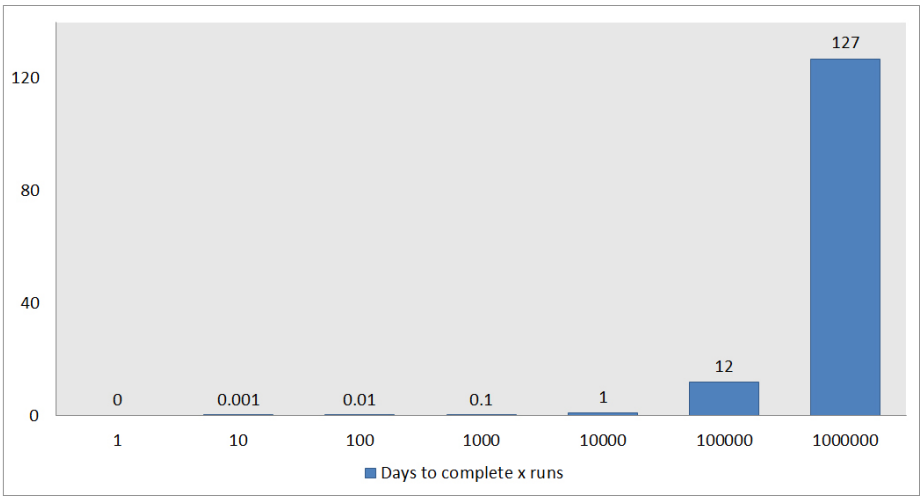

Figure 1.1: Amount of days required to complete various runs of the original simulation.

Departing from the MATLAB language allows us to profile, analyse and test the portability of the simulation, giving us a first step toward running the program on the reconfigurable hardware. Obtaining accurate outputs to the KMC method within an acceptable timeframe then allows us to simulate larger pieces of muscle with finer time steps. Since the results were a large volume of Force/Time pairs, we also discuss issues regarding disk accesses and Input/Output (I/O) bounds (chapter 5). The results will contribute to our understanding of how muscles work, with the benefits including potentially finding better ways of dealing with muscle problems.

1.3 Project Objectives & Achievements

In this section, we show that to meet the milestones and targets set above we were to follow various objectives. In the case of migrating the MATLAB code to a compiled language, we needed to first identify the performance gains we could make within the interpreted language. Since MATLAB was now serving as our pseudocode, it was in our best interests to rewrite the scripts as functions and refactor code which appeared slow or unnecessary. Once the KMC simulation in MATLAB was optimised to a saturation point, we were then faced with the design choice of selecting a language to migrate to. This included languages with intermediate code (such as Java with Java Bytecode, and C# with the Microsoft .NET Intermediate Language) or languages such as C and C++ with no such aspect.

Armed with the MATLAB as algorithmic pseudocode, the next step involved migrating the KMC simulation to the new language and checking for correct-

ness. Once we verified the program's outputs, we could look at performance gains and consider acceleration. For this, a number of choices were available, notably the Field Programmable Gate Array (FPGA) system. We present an evaluation of various methods available to us and our chosen means of acceleration, and demonstrate our achievement of obtaining the target number of runs comfortably under the target maximum of a day.

The following objectives were key in directing this project to its end and producing our contributions:

- Optimising the simulation in MATLAB (chapter 3). This was paramount in providing a base to begin migrating to a compiled language. We present optimisation methods including refactoring of code and performing translations to C++ to test speedup across various languages.
- An optimised version of the simulation in C++ (chapter 4). We maximise the performance gains available without acceleration by profiling the software and investigating what can be done to improve the slowest aspects of the code. We also add flexibility to the simulation by including a wrapper to allow it to be better suited to long term runs, with the ability to modify compliance values between runs for our experiment.
- An accelerated version of the simulation 2976 times faster than the original with greater accuracy (chapter 5). This enables us to see previously unmeasurable results. We draw on parallelism and acceleration to be able to perform a large amount of simulations in a reasonable timeframe.
- Analysis of the results which provides novel evidence of the dependence of crossbridges on compliance (chapter 6). We are able to show through analysis of the Force/Time relationships given by long term runs of the simulation that crossbridges appear to last longer in environments with smaller compliances. This has never been previously tested and we demonstrate various graphs which strongly suggest a relationship between compliance and maximum potential force.

1.4 Summary

In this chapter we looked at an overview of the main motivations for our project, notably the ability to understand more about a population of XB connections between myosin and actin filaments, fundamental structures within muscle cells. By learning more about how these attached myosin heads behave in relation to their neighbours, we can pave the way for future research into muscle activity. A Kinetic Monte Carlo method is used to simulate the states in which ten XB sites across the myosin filament transition between. We need to run this simulation for a very long time to obtain meaningful results, and our target is 1,000,000 runs. Under the current simulation, this would take four months of execution time to complete. We now realise the need for optimisation and acceleration

and target a maximum execution time of one day (127 faster than original projections) for all runs. This can only be done by migrating the simulation from MATLAB into a more optimisation friendly language such as C++. We then tailor the compiled program to match our various needs when simulating the muscle contraction process and allow for flexibility in its input parameters.

The following chapters present an optimised MATLAB code speeding up the simulation's execution time in chapter 3, a C++ version with extensive performance gains in chapter 4, and an acceleration method to speed up the simulation which performs 1,000,000 runs in under 3 hours (2976 times faster) in chapter 5, providing us with significantly greater performance and meets the target optimisation amount of 1 day (127 times faster).

Chapter 2

Background

In this chapter, we present the reader with an in-depth overview of muscle contraction and the reasons why this project helps make a contribution to our scientific understanding of the underlying processes. We look at the reasons why a simulation of such biological processes is time consuming and resource intensive and describe the methods of acceleration available along with their merits.

We also discuss the evolution of computing and processing power over time and how we can leverage these advancements in our work. We look into the Kinetic Monte Carlo mathematical model and its application in our simulation, and how it provides an abstraction from the complex biological process of muscle contraction whilst maintaining good accuracy in output.

2.1 Sliding Filament Model

In 1954, the Sliding Filament Model of muscle contraction was independently developed by both A. F. Huxley and R. Niedergerke and by H. Huxley and J. Hanson[1]. It is used to this date to describe the underlying system of muscles and how they contract.

Muscles can be categorised into those appearing along the skeletal foundation (striated muscles), on the arteries (smooth muscles) and within the heart (cardiac muscles). This project focuses on skeletal muscles, responsible for general mechanical motion in areas such as the arms and legs. Muscles can account for 40% of a human's body weight, and the longest muscles can have cells which are over a foot long. They can be seen to have a striated pattern. This is because a muscle consists of a group of many muscle fibres. Each muscle fibre is one elongated cell containing several nuclei, among the longest types of cell found in the body.

These muscle fibres, or myocytes, consist of myofibrils, a rod-like[4] unit composed of long proteins such as actin and myosin which are organised into thin and thick filaments. The actin and myosin filaments in striated muscle

are organised in a grid-like arrangement along the length of the myofibril. This arrangement is known as a sarcomere, the basic unit of a muscle.

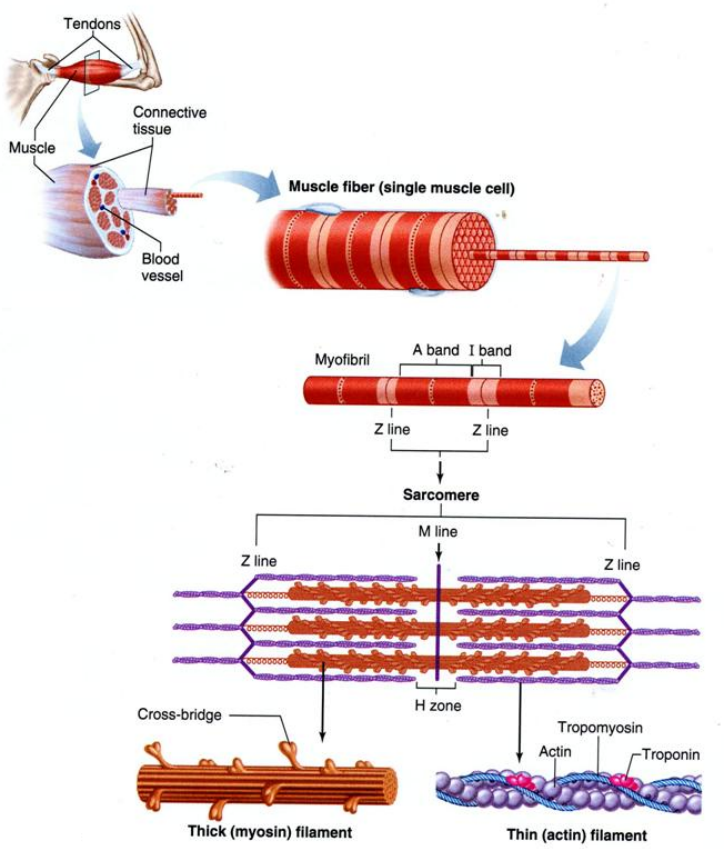

Figure 2.1: The arrangement of myofibrils in a muscle cell (Vander's Human Physiology).

Figure 2.1 shows an overview of how the fibres and cells form the resultant muscle. The sarcomeric arrangement accounts for the striated appearance of the filaments within the muscle fibres.

Muscle cells from biceps may contain over 100,000 sarcomeres each. The proteins within a sarcomere are seen as integral to muscle contraction and relaxation by the Sliding Filament Model. Figure 2.2 depicts a basic sarcomere cell. Actin, forming the thin filament, is shown above as binding to the borders of the sarcomere, known as the Z line, while myosin, forming the thick filament, is shown to have many ratchet-like heads (myosin heads) and consists of approximately 4000 amino acids.

The myosin filaments are attached to the end of the sarcomere by a pro-

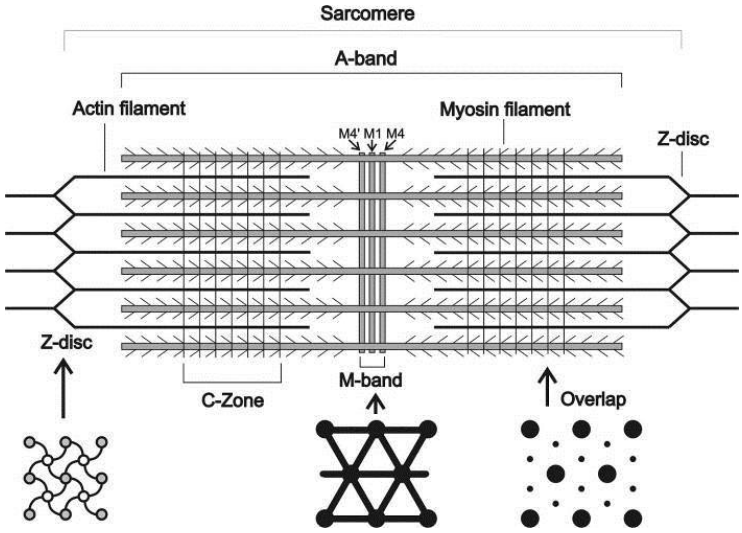

Figure 2.2: The sarcomeric region is composed of proteins and is integral to muscle contraction and relaxation.

tein called titin, and the actin filaments are surrounded by a protein known as tropomyosin. During muscle contraction, the myosin heads attach to binding sites on the actin filaments to form crossbridges (XBs).

These filaments are known to be compliant and the actin filaments on either end of the sarcomere slide due to XBs pulling them toward each other. Compliance is a measure of how much a fibre tends to resist recoil back to its original dimensions. Being compliant filaments, the muscle relaxes when the XBs detach and the filaments return to their original positions.

2.2 Muscle Contraction

The aforementioned Sliding Filament Model describes two filaments which are the major components of this system, the thin, actin filament and the thick, myosin filament. During muscle contraction, the muscle cells are stimulated by a signal known as an action potential. These are short lasting electrical events that are generated when the motor neuron stimulates the muscle fibre, causing the filaments to slide along one another, shortening the sarcomeres.

The Hodgkin-Huxley model[5] describes how action potentials in neurons propagate, treating excitable cells as biophysical entities with voltage gates and ion channels. The set of differential equations which approximate the electrical characteristics of these neuronal cells are well known as a result of this work. Action potentials initially cause calcium ions to be released from the sarcoplasmic reticulum. These ions force the tropomyosin surrounding the actin to displace, exposing the actin binding sites. Figure 2.3 shows the two filaments in detail.

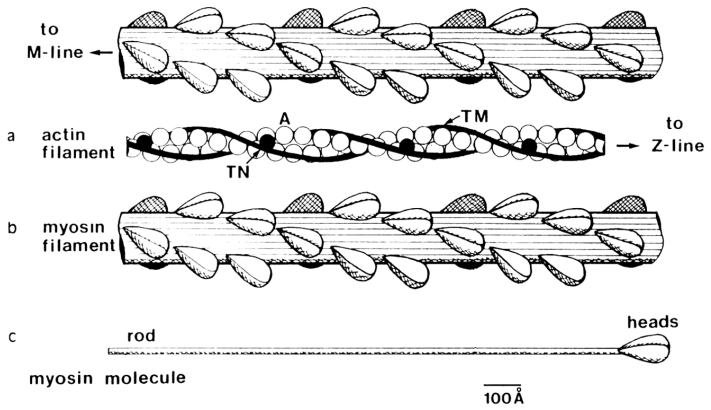

Figure 2.3: Actin and Myosin (thin and thick) filaments along with the myosin heads, troponin (TN) and tropomyosin (TM).

The myosin heads react to the calcium by moving and forming crossbridges (XBs) after attaching to the actin filament on its binding sites. The XBs then perform a movement known as a power stroke, sliding the actin filament across towards the center of the sarcomere, effectively shortening it for contraction.

Adenosine triphosphate (ATP), commonly considered the molecular unit of currency for energy transport within the body, is used to allow the XBs to detach after the power stroke. The ATP molecules are unhydrolized during this step to form adenosine diphosphate (ADP), and the myosin heads can reattach and perform another power stroke as XBs if calcium is still present. The actin typically does not return to its original position upon the detachment of one XB, since there are many XBs along the myosin filament attached to various binding sites on the actin preventing it from returning. If the ATP is also continually replaced by a new molecule, then the XB can detach repeatedly after its motions. These conditions cause the actin filament to be repeatedly pulled towards the centre of the sarcomere as the XBs detach, causing overall muscle contraction.

Many XBs may operate on actin filaments above and below the myosin. From this, we can derive two states the actin filament can be in: a relaxed state where the binding sites are blocked by tropomyosin, and an activated state where the presence of ATP causes the tropomyosin to move and reveal the binding sites.

The process is reversed for relaxation. When the action potential signals cease, the binding sites on the actin filament are once again blocked by the tropomyosin, causing the XBs to be unable to attach. The actin then returns to its original position. During isometric contraction, the attached XB has no force until the actin is undergoing filament sliding towards the M line at the centre of the sarcomere, reducing the H zone gap.

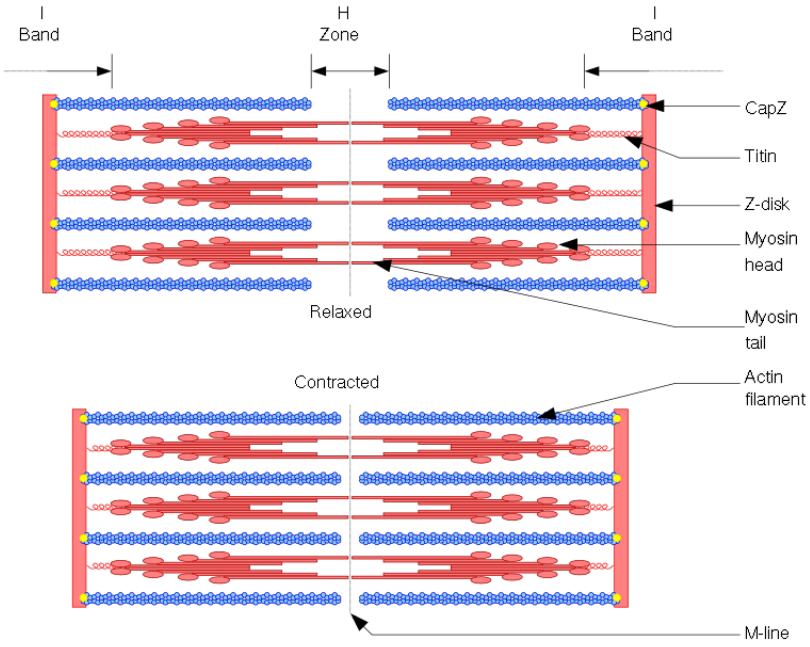

Figure 2.4: Relaxed and contracted states of a sarcomere structure.

When the XBs attach and cause the actin filaments to slide, the titin structures at the edges are stretched to extend from the Z-line of the sarcomere towards the M-line in the centre. Figure 2.4 depicts a sarcomere in both its relaxed and contracted states. The Sliding Filament Model predicts that the structural arrangement of all the filaments combined is unstable due to the forces acting on them[6]. This suggests that any initial imbalance would be detrimental to the structure of the sarcomere. Studies have, however, shown that the titin filament plays a role in the stability of myosin, and therefore the sarcomere[7].

2.3 Force–Velocity Relationship

It is possible to simulate the behaviour of one XB undergoing the cycle of attaching to an actin filament's binding site as a result of an action potential then detaching due to ATP usage. This is possible because the actomyosin interaction is believed to be stochastic. This simulation can then be scaled to include more than one XB, and studies have reproduced the same hyperbolic Force-Velocity relationship observed experimentally by way of simulating stochastic models[8].

A.F. Huxley's work on theories of muscle contraction[9] have shown that the force-velocity relationship relates to the fundamental steps causing the model's state transitions. We have learnt from his work and others that the inverse

relationship is affected by ATP and its unhydrolised form, ADP. Since the compliance values of the filaments involved are well known, we can represent the full process of muscle contraction as a mathematical model. This abstraction allows us to reduce the system to a set of states and probabilistic transitions between them.

2.4 Probabilistic Models

The simulation we use is a stochastic model of muscle contraction at the microscopic level. This uses the Kinetic Monte Carlo method of simulation, and is a more intuitive approach to the problem of determining whether XBs are independent or not compared to solving the differential equations to approximate a numerical solution. A long term simulation is more intuitive because it represents an actual neighbourhood of XBs interacting over a period of time. The model used is probabilistic since each state transition affects the chance of the next within the cycle. The problem of finding an algorithm to simulate the system at this level was posed by the medical supervisors to this project, R. C. Woledge and N. A. Curtin, in accordance with their research on muscles[10].

Monte Carlo algorithms are a class of randomized methods that solve problems in deterministic time using states and a degree of randomness within its logic. Monte Carlo algorithms are probabilistic in that they have a very small chance of producing incorrect results. This hurdle is overcome by running Monte Carlo algorithms for long periods of time, and simulating a system in such a manner can allow averages to be taken which represent the simulated system reasonably correctly. Our initial simulation uses a combination of algorithms which are very resource intensive, such as a matrix inversion. We now understand the need for optimisation.

2.4.1 Complexity Theory

Algorithms can be categorised in terms of their complexity in utilising time or space related resources. Decision problems that can be solved by polynomial-time Monte Carlo algorithms are classed as Bounded-error probabilistic polynomial (BPP). The error probability for results in this case is at most $1/3$.

We describe algorithms as having an order of complexity relative to the resources they utilise and the inputs. For instance, the Quicksort algorithm of sorting a list of n numbers has an average case performance of $O(nlogn)$ and a worst case performance of $O(n^2)$. This means that the efficiency of the algorithm depends on its inputs in that if the list was n large, the algorithm's worst case would require n^2 operations. The Big-O notation describes the bounding order of complexity, and we can understand these values to denote the Quicksort's worst case as having quadratic, or polynomial, time complexity.

Problems which can be solved by efficient algorithms of order n^2 are part of the complexity class P, denoting they can be solved in Deterministic Polynomial Time. Those which cannot, however, are classed as NP (Non-Deterministic

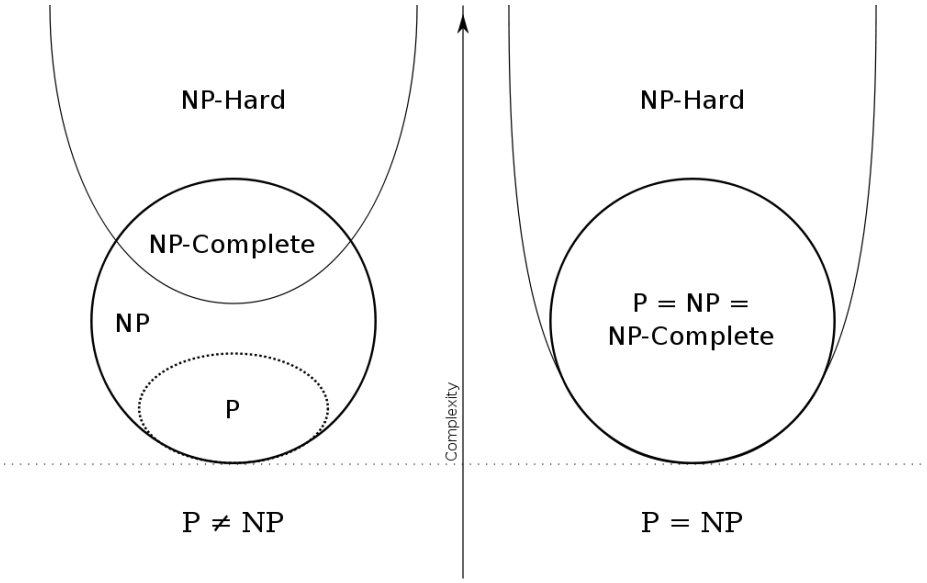

Figure 2.5: A breakdown of complexity classes for both outcomes of P vs NP.

Polynomial Time). The two complexity classes have not been proven to be different. Figure 2.5 shows a breakdown of the complexity classes according to the two possibilities. The challenge of proving whether or not $P = NP$ remains a Millennium Prize problem, touted the most important unsolved problem in Computer Science[11].

The problem revolves around the concept of whether a problem having quick machine verifiable solutions (P) means those solutions can also be found quickly. It is currently widely assumed that $P \neq NP$[12], and many proofs have been attempted to show either case[13]. A correct proof either way will have great impact, since the solution to P vs NP intrinsically links to solutions of the other Problems. If it can be shown that $P = NP$, then not only will a new era of cryptography need to be abruptly ushered in to accommodate the downfall of current security algorithms considered too difficult to brute-force, but NP-hard problems such as genome sequencing, protein structure prediction or muscle contraction simulation would become easier.

Algorithms can be rewritten to be less resource intensive, and with enough optimisation, we can downgrade its complexity class to that of a lower order for increased efficiency. A canonical example of this can be shown by reviewing the following code portraying a method of performing some_function on each value in a 2D array:

```
for (int i = 0; i < Xvalues; ++i) {
    for (int j=0; j < Yvalues; ++j) {
        some_function(array[i][j]);
```

```
    }
}
```

The amount of calls to some_function depend on the amount of items in the array, but since a double nested loop is used to traverse these items, the complexity of performing this action is $O(n^2)$. In the following code, however, we see a version that performs some_other_function on a 1D array, treating it as a list of elements. The amount of operations here simply depend on the length of the list, so we have a complexity of $O(n)$:

```
for (int i=0; i < array.length; i++)  {
    some_other_function(array[i]);
}
```

In our model, we look at methods to optimise and downgrade the complexity class of various algorithms used. We use the Kinetic Monte Carlo algorithm to perform our simulation of muscle contraction since it involves looking at the time evolution of the system. The algorithm is a probabilistic model with repeated random sampling to compute the states and XB forces. Previous studies have found correct methods to simulate XB activity whilst treating them as independent[14]. We look at the method used to simulate them as a dependent neighbourhood of interacting filaments and perform optimisations to be able to reliably measure the results in the long term.

2.5 Overview of Acceleration Methods

The speed at which a computer processor can process instructions depends on a variety of factors. At the most fundamental level, more transistors in a chip translate to greater performance. In 1965, G. E. Moore described a trend of transistors in integrated circuits doubling every year, now known as Moore's Law[15]. Figure 2.6 shows this trend over the course of 40 years using a vertical log scale.

Moore's Law is theorised to continue steadily until limits regarding transfer rates within a processor are reached. For instance, the problem of heat caused by resistance in a circuit with many transistors becomes an issue as their numbers increase. One way this limit has been overcome has been by introducing the notion of multi-core processors. With many cores, Moore's Law has been able to continue progressing steadily, though developers have had to take advantage of parallelism and threading in order to speed up their algorithms. In addition to CPUs, Graphical Processing Units (GPUs) are increasing in speed steadily over time. These are special purpose processors which can handle floating point operations (FLOPS) very fast, making them ideal for use in graphical applications such as games or simulations which requires massive amounts of numerical manipulation.

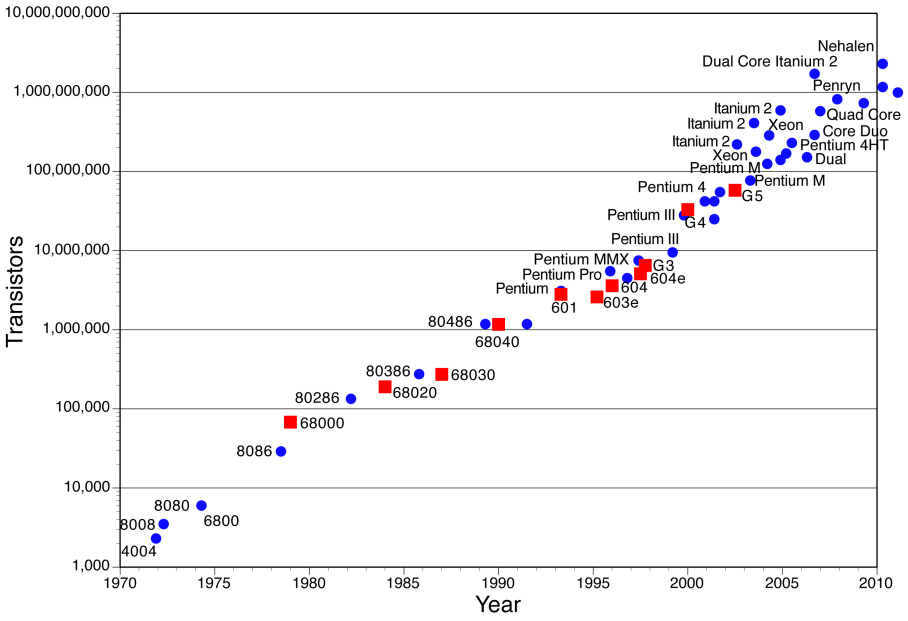

Figure 2.6: Number of transistors in various processors by year. The exponential growth corresponds to Moore's Law.

2.5.1 Acceleration

We can only optimise algorithms in software to a maximum limit to take advantage of the hardware being run on. There comes a point at which improving the hardware becomes the easiest way to make performance gains. At this point, we must make decisions on how to make the biggest leaps in speed for our software. There are a number of advantages to using different types of acceleration, notably the following.

- Increased throughput by way of massively parallel hardware.
- Decreased latency for IO operations.
- Faster execution of loops and matrix inversions.
- Obtaining more accurate outputs which weren't previously possible.

In this section, we look at various methods of acceleration and why they are considered in reimplementing our simulation.

2.5.2 Reconfigurable Hardware

We assume the reader has a good background in computing, but since this is a specialist subject, we will provide some information. Reconfigurable hardware

systems allow developers to harness the high performance aspects of a hardware implementation while not fixing the system entirely during use. The hardware's data paths can be changed to suit the development needs of the user. FPGAs are a common reconfigurable hardware architecture used today, consisting of arrays of extremely fast logic blocks which can be rewired and combined to perform logical operations.

Reconfigurable hardware operates by allowing the data flow of a program to change as well as the control flow. This, coupled with the ability to adapt at runtime by changing the circuit on the reconfigurable board, allows this custom computing method to have many gains over traditional means for acceleration. G. Estrin first proposed the concept of being able to use reconfigurable hardware[16] with a typical control flow processor serving as the system's master, allowing it to control the overall behaviour. This opened up the ability to use these hybrid structures as dedicated hardware for specific software tasks. They were soon to become the key to optimisation.

Since reconfigurable hardware use slower clock rates and lower specifications than typical microprocessors, it seems infeasible that they would perform magnitudes faster on some tasks. We now discuss why this is the case for such dataflow computing paradigms.

Dataflow vs Control Flow

Traditional microprocessors operating under the control flow paradigm, i.e. a fetch, decode, execute cycle, can be considered analogous to a set of production line workers in a factory who each have expertise in a variety of tasks. When one worker receives output from the previous worker, it can determine the next step towards obtaining the output and carry this out. This lineup of 'general experts' is expensive and slow compared to the analogy applied to dataflow cores. In dataflow, each worker is able to perform only one function very efficiently. If lined up in the correct way, the route from inputs to outputs can be realised exceptionally fast compared to control flow.

In a dataflow environment, an input stream is picked and considered the "flow", with dataflow cores performing specific functions in the order necessary for providing the right output. As processors introduce more and more simple cores on the same chip to sustain Moore's law and continually overcome its limitations, dataflow engines take this concept to the extreme.

Figure 2.7 shows a control flow system incorporating the fetch, decode, execute cycle. The processor holds internal values such as the Program Counter, which points to the next instruction to be processed, and registers are checked for data and instructions to operate with. This method of performing a large software task can mean many CPU cycles are required to perform a single sub-operation to obtain intermediate outputs. For repeated tasks such as loops, it may mean that one iteration of a loop may take significantly longer than one CPU cycle.

Figure 2.8 depicts a dataflow system, in which a chip is made up of many simple cores that may be configured to perform one function very quickly. If the

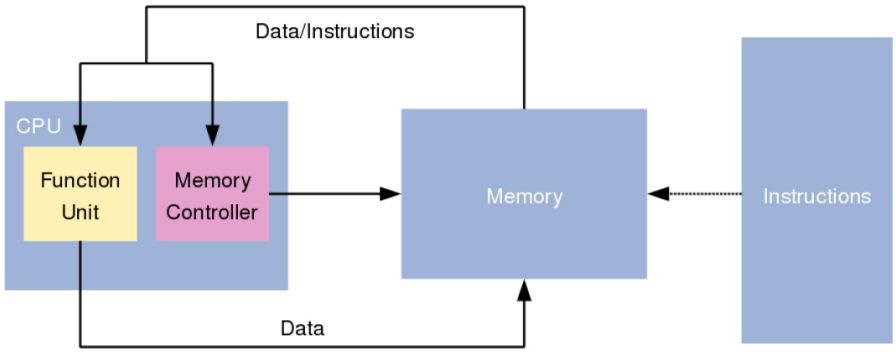

Figure 2.7: A control flow system (Courtesy of Maxeler).

contents of a software loop are translated by way of configuring a dataflow path using such cores, every iteration of the loop will complete on one clock cycle. The potential for massive parallelism provided by dataflow engines offsets the compromise of having a slower clock rate.

Feature	Performance Change
Slower clock rate	-10x
Greater degree of architectural freedom	+100x
Bit level parallelism	+100x
Pipeline level parallelism	+100x
Architecture level parallelism	+100x
System level parallelism	+100x

Table 2.1: Acceleration potential of dataflow engines over control flow systems.

Table 2.1 explores the acceleration potential of dataflow engines and their relative performance gains. From this, it becomes clear that the more complex a software problem is, the better the performance gains from dataflow will be over control flow. This is why dataflow engines (DFEs) are considered for game-changing scenarios where optimisation and other acceleration methods are insufficient. There are two types of major custom computing hardware available, Application Specific Integrated Circuits (ASICs), which are typically not intended for general purpose use, and Field Programmable Gate Arrays (FPGAs), which are a more cost-effective equivalent that allow easy reconfiguration on the fly.

Deciding on the input stream to use as the dataflow can be difficult. Acceleration by FPGAs remains a niche and often last resort due to the amount of optimisation and refactoring of existing code required in order to simply run a program on the hardware. When this is possible, however, the complexity of a problem is challenged by the paradigm itself. For instance, the following code

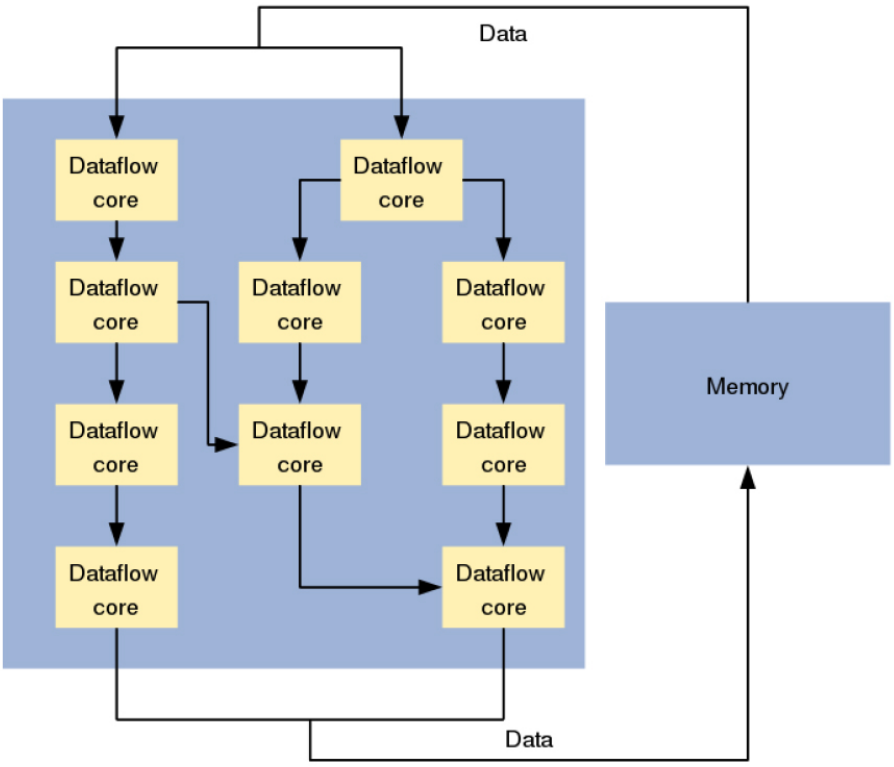

Figure 2.8: A dataflow system (Courtesy of Maxeler).

in a control flow environment would be executed iteration by iteration on every element of an array.

```
for (int i=0; i < array.length; i++)  {
    output[i] = process(array[i]);
}
```

In a dataflow environment, however, we can identify the elements of array as our input stream, therefore eliminating one loop from having to be hardware implemented. The elements of array would be passed in and on every clock cycle, the previous element will have returned an output after processing. This means that the first iteration may have some overhead, but once the system is running for some time, future values will be greatly accelerated compared to control flow.

Figure 2.9 shows the performance gains from using DFEs as opposed to control flow microprocessors for 3D finite difference modelling. The notion that the more complex a problem is, the greater the performance gains will be becomes

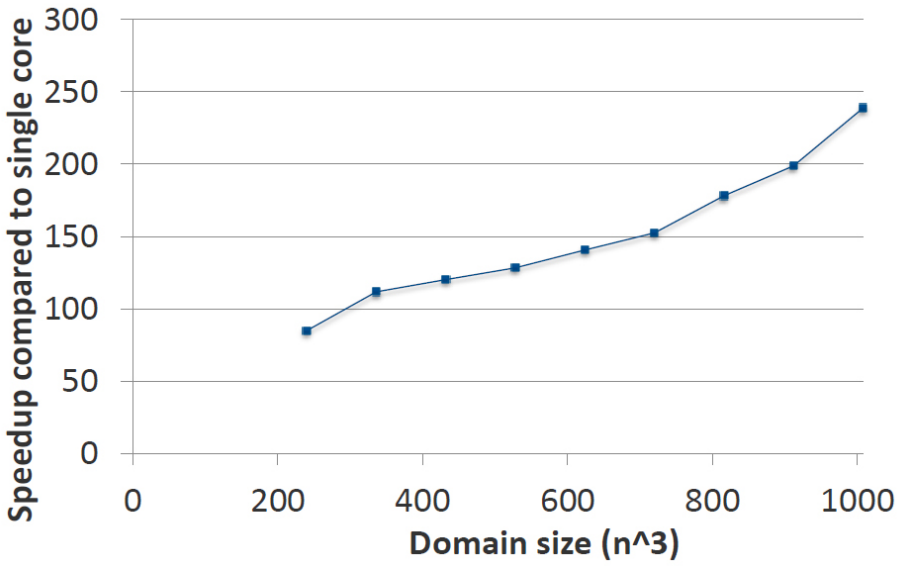

Figure 2.9: 3D Finite Difference Modelling relative speedup by DFEs (Maxeler).

clear upon realisation that traditional CPUs will not be able to parallelise the software program as much. We only discuss FPGAs as a potential acceleration option for our simulation due to their ability to implement any logical function ASICs can perform but with cheaper hardware and the ability to be customised after manufacturing.

Field Programmable Gate Arrays

FPGAs are a special case of integrated circuit which allow reconfiguration after the chip's manufacturing. These typically serve as the dataflow engine (DFE) in commercial solutions which provide hardware acceleration as a service. The FPGAs have on board memory as well as many dataflow cores, the behaviour of which is described by a Hardware Description Language (HDL) specification. Automated techniques are then applied to generate the electronic design by mapping the connectivity of the circuit by way of a netlist. The netlist is then fitted to the FPGA architecture using a process called place and route.

The place and route process first triggers a placement step, determining where all the required logic elements and circuitry should be placed in the limited space available on the FPGA. This occurs on the grid of the FPGA and is essential in configuring the dataflow circuitry. When placement is complete and the locations of the electronic components are fixed, a routing step follows, in which the design for connecting wires between placed components is determined. Once the mapping, placing and routing is complete, the FPGA is configured to run the desired dataflow program.

These steps are typically performed by an automated program after designing the intended FPGA circuit using logic diagrams. These diagrams may contain a type of HDL known as VHDL, often likened to an assembly language for dataflow.

VHDL

VHSIC (Very High Speed Integrated Circuit) Hardware Definition Languages (VHDLs) were initially developed by the US Department of Defence to describe how ASICs behaved. Simulators are able to take VHDL files as input and synthesise them into circuit implementation details for use in the map, place and route processes. VHDLs contain native constructs for parallelism in various hardware, and it has since come into use for FPGAs.

The following code implements an OR gate in VHDL.

```
entity ORGATE is
  port (
      I1 : in bit;
      I2 : in bit;
      O1 : out bit
       );
end entity ORGATE;

architecture RTL of ORGATE is
begin
    O1 <= I1 or I2;
end architecture RTL;
```

The inputs, outputs and abstract Register Transfer Level (RTL) architecture are described in the language, allowing this to be mapped and wired correctly during the map, place and route processes.

Rather than directly using a VHDL, developers typically use a high level description for running code on the low level hardware. Notable examples include Handel-C, a rich subset of the C language, or MaxJava from Maxeler. The Maxeler platform consists of both the hardware and software packages required for the efficient implementation of algorithms on an FPGA system.

Maxeler Platform

The Maxeler platform describes both the hardware FPGA packages offered by the company along with the MaxCompiler Integrated Development Environment (IDE) which uses their implementation of Java to describe the dataflow graph for the FPGA. The Maxeler hardware is available in two forms, MaxNodes and MaxWorkstations. The MaxNodes can be combined into a server arrangement to form a MaxRack.

Table 2.2 shows some hardware arrangements of the Maxeler hardware. A MaxRack can contain many MaxNodes, each with up to four MaxCards and a

Hardware	Dataflow Engine	CPU	RAM
MaxCard	1	-	24 – 48GB
MaxNode	4 (1U)	Intel Xeon	192GB
MaxRack	Many (10/20/40U)	Many	Many

Table 2.2: A partial lineup of dataflow hardware offered by Maxeler.

maximum of 48GB RAM each. These are coupled with Intel Xeon cores and a large shared RAM connected via a high speed MaxRing interconnect. The CPUs are designed to serve as the master devices to the DFE slaves. The FPGAs are produced by Xilinx and the hardware runs the CentOS Linux distribution, using a variant of Java, MaxJava (MaxJ), as its VHDL. A C++ library called MaxRT is also available to allow developer machines the ability to interact with the FPGAs.

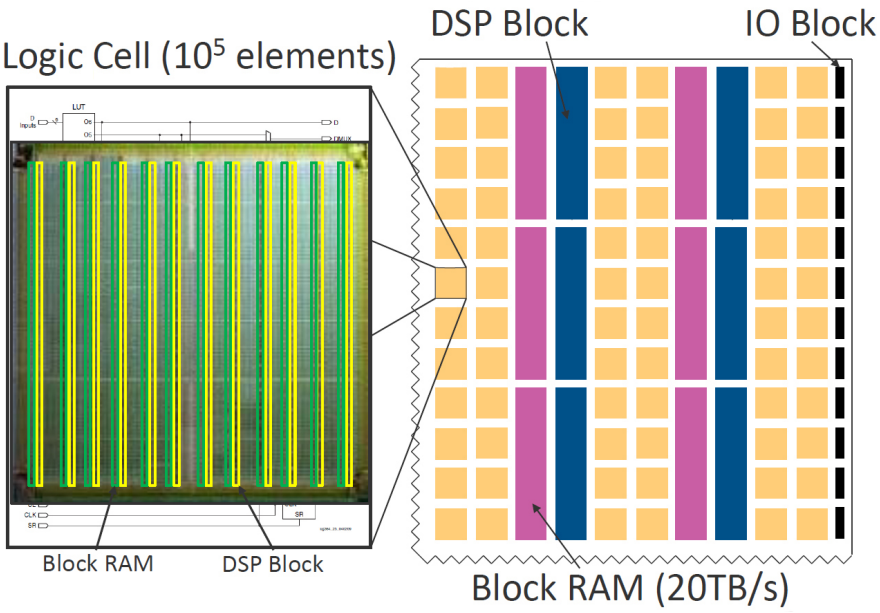

Figure 2.10: A dataflow chip (Courtesy of Maxeler).

An example dataflow engine is shown in Figure 2.10. The chip can be seen to composed almost entirely of logic cores, as opposed to CPUs with various other components. Aspects relating to the operating system, MaxelerOS, border the chip on its perimeter. This allows the chip's space to be used to its full potential. Developers who take software and attempt to target this FPGA technology for hardware acceleration must refactor the code to work with the MaxCompiler.

The MaxCompiler IDE is a modified version of the Eclipse IDE, with a modified version of Java serving as the descriptive code for the dataflow.

It is important to note that developers using this method typically aim to accelerate the slowest portion of their software, first isolating it in a MaxJ kernel which defines the dataflow graph for the operations, then linking the FPGA accelerated part to the original hostcode. This means the hostcode needs to be modified to be able to handle communication with the dataflow system, and when ready, it will pass input to the DFE as an input stream, expecting it to report back with the optimised outputs.

MaxJava & Hardware Variables

Although MaxJ is an extension of Java, it should not be treated as a language residing in the control flow paradigm. MaxJ serves the single purpose of defining the dataflow graph for the FPGA to implement. This results in the introduction of custom types for use with the FPGA. One such example is the HWVar type. The following code represents a simple kernel which would perform an increment on the input stream.

```
HWVar x = io.input('x', hwInt32);
HWVar y = x + 1;
io.output('y', y, hwInt32);
```

This MaxJ represents the graph shown in Figure 2.11, since the inputs have a constant value of 1 added. Graphs from MaxJ become complex very quickly.

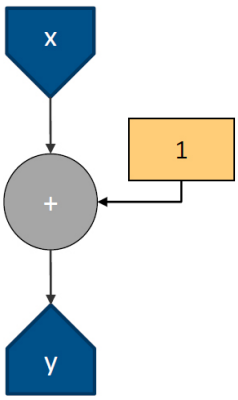

Figure 2.11: Dataflow graph for an increment function (Courtesy of Maxeler).

The following code is another example of how MaxJ translates into the underlying graph.

```
HWVar x = io.input('x', hwInt32);
HWVar y = x + x + x;
```

```
io.output('y', y, hwInt32);
```

Figure 2.12 shows how the three values are fed into each operation to result in the desired $x + x + x$ output.

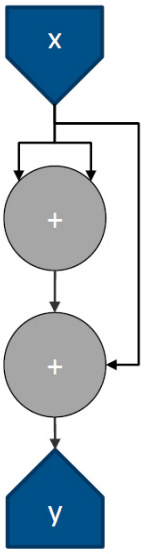

Figure 2.12: Dataflow graph for simple addition function (Courtesy of Maxeler).

From this, it becomes apparent that the HWVar type is not to be treated as a static value and that each line in the MaxJ code represents a transition from one operation in the dataflow graph to another. Furthermore, it is evident that use of the HWVar type roughly translates to parts of the data stream, and any pure Java aspects inside the MaxJ code will not be included in the graph. This renders the following operations illegal under MaxJ.

- Using the value of a Java variable in a conditional.
- Assigning an HWVar to a Java variable such as int.

Furthermore, since loops are a control flow structure and are pure Java, they are either fully unrolled (should resources permit) or partially unrolled to determine the full dataflow graph before execution. This allows loops to provide a shorthand way of describing repeated functions within the graph.

The MaxJ libraries provide various additional functions to take advantage of dataflow programming concepts, such as using a streamOffset in order to maintain future or past values for repeated use. This is particularly useful for performing a moving average of various values within an input stream.

We look into methods of leveraging the MaxCompiler tools for accelerating our simulation in chapter 5. We approach this problem by profiling the compiled version of the simulation and optimising the slowest functions for acceleration. The FPGA hardware, however, is considered very niche and other forms of acceleration are available for consideration. One such option involves running the simulation on GPGPUs.

2.5.3 General Purpose Graphics Processing Units

Graphics Processing Units (GPUs) are specialised processors that use most of their transistors for floating point arithmetic. This allows them to perform tasks related to graphical display in a superior manner to typical CPUs. GPUs are therefore used to handle intensive graphical elements in software, such as physics engines in games.

The action of attempting to perform computation on GPUs which would otherwise be handled by CPUs has become increasingly common. For this, we look at General Purpose Computation on Graphics Processing Units (GPGPUs). Due to their parallel nature, GPUs seemingly appear to be an ideal and cost effective solution to any numeric based problem requiring acceleration. However, this is not the case. The programmability of GPUs was initially limited and developers looked into methods of disguising their general programming problems as numeric problems of matrix and number form.

The increasing demand for GPGPUs saw the eventual onset of API development and support, and in 2007, NVIDIA released their Compute Unified Device Architecture (CUDA). With CUDA code, developers are now able to parallelise their problems over a set of GPUs, and there have been many supercomputer class machines built to take advantage of this architecture[3].

The use of GPUs in performing floating point and arithmetic calculations in academic work and other simulations have allowed limitations from CPUs to be significantly overcome.

Figure 2.13 shows different CPU and GPU costs along with the time it takes for them to relatively perform arithmetic calculations. It becomes apparent that the GPU is both the more economic solution for accelerating a problem with numerical intensiveness and the faster choice.

2.5.4 High Throughput Computing

High Throughput Computing (HTC) is the process of using computing resources to perform a large computational task over a long period of time. While High Performance Computing (HPC) looks at using large amounts of compute power for short periods of time to execute a task, HTC concerns long term usage such as months and years. This can be seen in various "at home" projects, including the Great Internet Mersenne Prime Search (GIMPS), which distributes the difficult computing problem of finding the next Mersenne prime (a prime number that is one less than a power of two such that $M_p = 2^p - 1$). The throughput of GIMPS at the time of writing is measured at approximately 86 teraflops as of March

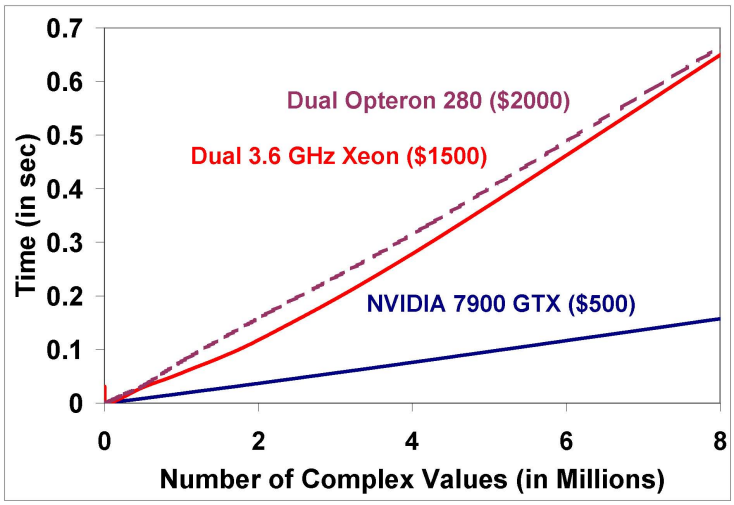

Figure 2.13: Arithmetic calculations over time between GPUs and CPUs

2012. While HPC providers tend to measure their success in FLOPS, scientists using HTC are more concerned with how many floating point operations can be extracted per month or year. Grid computing with systems such as Condor has seen prolific use in the scientific community[18].

The advantages of distributed HTC are vast, and notably include the following.

- The use of large scale parallelism across machines.
- The system remains robust if some nodes are switched off or crash.
- Its cycle stealing nature preventing intrusiveness, only idle nodes are utilised.
- The ability to add or remove nodes to improve the system easily.
- Running a large computation across a wide variety of devices connected through a client.

Condor

The Department of Computing, Imperial College, has a Condor HTC system installed on their lab machines. Condor is an HTC framework developed by the University of Wisconsin-Madison's Computer Sciences Department, and involves making a distributed high performance grid from unused resources in a network of machines. Since the Department of Computing has periods of small use, such as during the night, the Condor pool of available resources increases.

There are over 200 machines available, and each available core is considered a node in the Condor network. The typical machine specifications include four

Intel Core i5 650 Dual Core CPUs clocked at 3.2GHz. Since this provides each machine with 8 logical cores, and there are over 200 machines, we can see that the Condor installation provides parallelism across 1600 individual nodes. From our production runs in chapter 5 it becomes apparent that limitations due to busy lab periods and software caps allow approximately 250 simultaneous runs.

The Condor system operates in a cycle stealing mode, utilising resources on available machines which are otherwise unused by anyone else. This means the Condor system will provide the highest throughput when machines are not subjected to any physical use. Condor integrates with devices across a range of operating systems and uses a daemon which accepts jobs. A Condor job is submitted by way of a script which describes an executable file to be run. This script can specify the quantity of jobs required and any command line arguments which are needed. All output written to stdout or stderr by the executable is saved as a unique file corresponding to the job number in a shared space that all Condor nodes can access.

We look into ways of performing acceleration using Condor to simulate a High Performance Computing system. The Condor system schedules tasks of certain estimated run times more often. We use this information to form jobs which take advantage of the scheduler. These aspects are discussed in later chapters.

2.6 Summary

In this chapter we have presented a background and introduction to the biological motivations behind this project from understanding the Sliding Filament Model of muscle contraction to why we need to simulate crossbridge activity at such a fine level. From information surrounding the biological aspects presented here, including the Force-Velocity relationship, it becomes clear that building a computational model which simulates the process presents a more intuitive solution than solving a set of complex differential equations representing the system analytically.

We present background regarding computational models, and the importance of the Kinetic Monte Carlo method to our simulation is demonstrated. The original simulation is explained in an in-depth manner to inform the reader of the steps used to ensure correctness in the model, and from this we give theory which provides us the means to optimise and accelerate the simulation to reach our target number of runs and find out more about the muscle contraction process. For this, we explain computational complexity theory and what it means for an algorithm to be categorised in a complexity class, as well as how to downgrade them and make algorithms more efficient.

We also introduce the reader to a basic history of computing advances related to Moore's Law and explain the current state-of-the-art in acceleration. This includes the options of GPGPU acceleration or reconfigurable hardware, the latter leading us into discussing the benefits and methods of FPGA acceleration. We look at dataflow programming as a direct contrast to control flow,

and discuss how we can implement the simulation on the FPGA system using the Maxeler specialist tools. This leads to a description of alternative High Throughput Methods available and the potential to harness massive parallelism without the need to reimplement a solution in a custom language.

Throughout this chapter we present current research relating to muscle contraction. Although no research has yet calculated the long term forces in the manner of this project, we emphasise the necessity to do so in order to better understand the contraction process and demonstrate not only the ability to make these calculations possible by way of optimisation, but an interpretation of the data itself. We therefore present novel results in this project and look at methods of analysis to be able to explain them as future work.

Chapter 3

Optimising the KMC Simulation in MATLAB

In this chapter we present a refactoring of the KMC simulation code with software optimisations that allow performance gains in MATLAB and a departure from the MATLAB code to a compiled language. The original MATLAB code is initially written as a script. Scripts are interpreted and run from the MATLAB IDE, and it is generally assumed that functions perform faster than scripts on average. We therefore take first steps in converting the script to a function by adding the following code to the header of the KMCmany.m script. Before we can present our conversions, we must first understand how the simulation is an accurate representation of the underlying biological process.

3.1 The KMC Simulation

The simulation with which we begin is a MATLAB script which performs a KMC simulation in this manner for a population of ten XBs. The advantage of performing the simulation at this level of detail is that the interaction between filaments can be taken into account and the method is representative of the true underlying biological processes. If we assume the XBs do not interact with each other, the simulation for a population of ten is a trivial upscaling of previous work which looked at only one XB. This is because the probability of each reaction would depend on the state of one XB, with values corresponding to those in a Look Up Table (LUT) built from well known values from prior experiments.

When considering the interactions between XBs, the number of states is too large for a LUT and the state must be recalculated when each XB changes. We present a MATLAB script for finding the XB states and forces from the filament dimensions and compliances. The equations to be solved for calculating the states are represented internally as a matrix data type. Each iteration of the KMC method performs a matrix inversion and we show that this method and

all future transformations produce correct results.

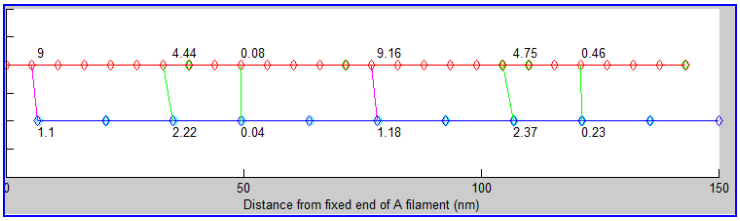

Figure 3.1: Actin (red) and myosin (blue) filaments, binding sites (circles) and connected XBs (vertical connecting lines) within the KMC simulation.

Simulating many XBs individually allows identification of events which cause states to change with finer granularity, as well as providing greater accuracy in the outputs. Our simulation entails the following example system shown in Figure 3.1.

The number above each XB shown represents the force in pico-Newtons (pN) with the numbers below each XB representing the distance between the two ends in nanometres (nm). This represents a one dimensional problem outlining nodes within a system which can either be XBs, myosin heads or actin binding sites. Only some of the XBs are present in the above diagram, with the actin filament consisting of 26 nodes, each having an unstressed length of 5.5nm. The myosin filament is fixed at 150nm and consists of 10 nodes (myosin heads, potential XBs), each of which has an unstressed length of 14.3nm. The two filaments are joined by a number of XBs extending from the myosin to the actin filament, each having a length representing the distance between two connected nodes. The unstressed value of this length can be one of three values depending on the state it is in. These are 0nm (relaxed), -3.4nm and 6.8nm.

Since the model looks at the time evolution of the muscle contraction process, the kinetic states of the system translate to the four states in the reaction scheme of each XB site. With ten simulated XB sites there are 4^{10} possible states. The evolution of the states, and therefore the resulting physical effects on the XBs from the neighbouring movement provides a more accurate way of observing the XB activity since it takes into account each XB alternatively rather than approximating a population.

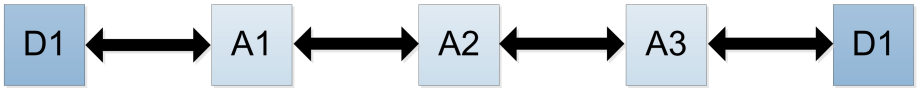

Figure 3.2: The reaction scheme showing kinetic states of the simulated system.

From this information we know the kinetic states of the system. Each XB site can be in one of the four states in the reaction scheme shown in Figure 3.2. A1 – A3 are states in which the XBs are attached and D1 is a detached state. A LUT containing the reaction rates in the form of an 8x20001 matrix is read

by the KMC simulation. These are well known values derived from both consistent experimental observation and theory. From this, we know the compliances (stretching) of the two filaments and the list of XBs for the connected sites along with their states, hence we are able to calculate the force within the system, the position of each node and the length of each XB. This allows us to then determine the probability of further state transitions, thus simulating the evolving system over time.

-0	-0	-0	...
7000	7000	7000	...
7000	7000	7000	...
523.4	523.4	523.4	...
-0	-0	-0	...
19.498	19.509	19.521	...
77.622	77.667	77.713	...
0.00049903	0.0004993	0.00049958	...

Table 3.1: An 8x3 sample of the 8x20001 matrix used in the KMC simulation.

Table 3.1 displays a sample of the LUT values used within the simulation. Each XB in the simulation begins detached in state D1 and undergoes a transition to the attachment state A1. This represents the myosin head attaching to the actin filament, after which we represent the full cycle of muscle contraction as the transition between states A1 – A3. We assume that every XB is an entire myosin molecule and only one myosin head can attach. We model the behaviour of XBs over these transitions with random time steps up to a limit (finishing time) of 0.1s. The outputs to one run of the simulation are initially set to output up to 100 Force/Time pairs. This is later changed to save up to 5000 for accuracy purposes. This precisely follows the generic (KMC) algorithm using the following steps.

- Initialise a time $t = 0$ and establish a finishing time for the simulation T.
- Establish a list of rates r_i for each transition W_i in the system.
- Calculate a cumulative function $R_i = \sum_{j=0}^{i} r_j$ for all transitions $i = 1..N$.
- Calculate a random number in a uniform manner such that $n \in [0, 1]$.
- Determine which event i occurs such that $R_{i-1} < nR \leq R_i$.
- Change the state of the system as a result of the event and re-evaluate all W_i and r_i.
- Calculate a new uniform random number $n \in [0, 1]$.
- Update t by calculating $t = t + \Delta t$ such that $\Delta t = -log(n)/R$.

- If $t > T$, stop. Otherwise, perform all the above steps in order again.

The advantage of using this method over other algorithms is that we can use our pre-existing knowledge (the LUT) to reduce the amount of computation required. KMC algorithms only deal with rates, and this is an ideal way of modelling our desired behaviour due to its evolution over randomised time steps. The simulation consists of three parts: a KMCmany.m script which serves as an entry point to perform the KMC reactions, a GFFunc.m script which takes as input the system's states and returns as output the respective forces by way of a matrix inversion, and a Make_KMC_LUT.m script for generating the input LUT.

3.2 Analysing the MATLAB script

In this section we present three key optimisations to the MATLAB script which are instrumental in preparing the simulation for an easier transition into a compiled language. We look at the following to improve the MATLAB script.

- Converting the script to a function.
- Preallocating large variables for speed.
- Preparing the code for translation by refactoring required methods.

We first present to the reader an initial profiling of the KMC simulation in its unchanged MATLAB script format (shown in Table 3.2). This shows that approximately 60% of processing time involves performing the matrix inversion function call in GFfunc.

Function Name	Calls	Total Time	Self Time
KMCmany	1	5.817s	1.792s
GFfunc	4128	3.807s	3.807s

Table 3.2: MATLAB profile of the simulation as a script.

We begin by converting the script to a function. It is evident that the function has no inputs, but will return two matrix variables for the Time and Force respectively. We add the following header to ensure the code is not treated as a script by MATLAB.

```
function [OutputTime, OutputForce] = KMCmanyfunc
```

To make further improvements to the existing code, we identify key areas where large matrix variables are forced to dynamically expand and rewrite them by preallocating the matrix with their known maximum size[17]. For example, if a loop expected to run for a long time contains assignments to a variable not previously declared, it will need to expand on each iteration of the loop to account for the additional space used. This step involves the matrix being

copied to a memory location of a larger size to be able to accommodate the increase. The following code gives an example of dynamic expansion to create a 1x1000000 matrix of ones.

```
for i = 1:1000000
    Matrix(i) = 1;
end
```

In this case the "Matrix" variable is not preallocated and MATLAB would need to interrupt the iterations at various points in order to increase the size accordingly. A fix would be to preallocate Matrix as in the following first.

```
Matrix = zeros(1,1000000);
for i = 1:1000000
    Matrix(i) = 1;
end
```

Though this increases performance by letting MATLAB know the size we intend Matrix to be, it should be noted that this is still not yet fully optimised. The zeros function creates and initialises a matrix of the specified size while setting each value to 0. An equivalent of this function exists to perform the above without explicitly requiring a loop, thus reducing this code to the following.

```
Matrix = ones(1,1000000);
```

By replacing various instances of this within both the new KMCmanyfunc.m implementation of KMCmany.m file and the GFfunc.m files, we can be sure that MATLAB can preallocate space for matrix variables in large loops.

Function Name	Calls	Total Time	Self Time
KMCmany	1	4.913s	1.71s
GFfunc	3992	3.541s	3.541s

Table 3.3: MATLAB profile of the simulation as a partially optimised function.

Table 3.3 displays the profiled version after these transformations in the largest loops within the code. The performance gains from the previous version are no more than one second. Clearly, writing a wrapper around the MATLAB code, script or function, will not provide enough time optimisation to obtain the results from a million runs in a feasible time.

We now begin to look into translating the code and migrating to a compiled language potentially offering faster array operations and better performance on our target machine.

3.3 Programming Language Choices

In this section we discuss whether to use C# or C++ as the language to rewrite the simulation code in. Since C# is a language which runs on the .NET runtime, debugging tools are very powerful and some language features provide

an advantage over C++. These include allowing only safe implicit type conversions, such as integer widening to prevent data loss and the requirement to mark user-defined conversions as explicit or implicit, rather than the C++ approach of assuming implicitness.

Like Java, C# compiles into a machine and language independent intermediate language (IL) to run in a managed execution environment under the .NET Framework. Though this adds time to the compilation, it allows for a increased compile-time error checking and powerful code reflection capabilities. C++ on the other hand, like C, would take time to create any objects, compile and link the program files into a native executable. Targeting C++ would allow the KMC simulation to run on both Windows or Unix (Linux & Mac OS X) environments, an advantage for ensuring maximum throughput when submitting to the Condor network.

This portability along with the discovery of a built in MATLAB feature for automating and assisting a code translation to C/C++ led to our decision in targeting C++ as the language of choice. The feature in question is a tool within Simulink, known as Coder.

3.4 Simulink Coder

Simulink is a tool for modelling and analysing dynamic systems. The Simulink Coder can automatically generate C or C++ source for realtime implementation of MATLAB functions. The HDL Coder can perform automatic generation of VHDL code for FPGAs, and both methods perform systematic validation and verification to ensure model design errors such as overflow and division by zero are not permitted to occur. A subset of the existing built in MATLAB functions can also be automatically translated to C++ counterparts.

To enable use of Coder for translating our KMC simulation into C++, we added the following line to both the KMCmanyfunc.m and GFfunc.m code. This caused MATLAB to show warnings and errors specific to code generation.

```
%#codegen
```

Coder, after translating the MATLAB entry point code along with its inputs, also translates and links any dependent files. As a result, our entry point was KMCmanyfunc.m. It also requires providing a main.cpp as the C++ entry point.

The code analyser requires the following to hold in order to achieve correct automated code generation.

- Variables must be fully defined before they are subscripted.
- Variable sizes may not grow through indexing.

As a result, we needed to preallocate the sizes of every matrix in the code instead of only those contained in large loops.

3.4.1 Correctness

Since the MATLAB built in load function was not available for code generation, we substituted the line which loads the LUT from its separate file and included the entire LUT generation code. Centralising this into the simulation to create a self contained version would allow us to later prevent having to make many read requests to a large file in environments such as Condor. For the purposes of the MATLAB version, however, it would initially appear to be a step to a more inefficient design. This is later shown not to be the case by way of preventing the generation from occurring more than once.

Function Name	Calls	Total Time	Self Time
KMCmany	1	6.41s	2.082s
GFfunc	4429	4.023s	4.023s

Table 3.4: MATLAB profile of the simulation with LUT generation.

Table 3.4 demonstrates the performance hit taken when including the LUT generation code which was necessary to allow automatic code generation to complete. Relative approximate slowdown of an extra second occurred by introducing this step.

At this point, the research benefits of having this simulation running fast in C++ were realised, and to ensure we could comfortably analyse the results, we modified the output of the simulation to provide 5000 time points instead of the original 100. This means that for the target 1,000,000 runs, we would need 5,000,000,000 force/time pairs to be stored for analysis.

Using this new criteria and profiling the MATLAB result after modifying the simulation to provide these time points, we saw an order of magnitude slowdown in the simulation's performance, as shown in Table 3.5.

Function Name	Calls	Total Time	Self Time
KMCmany	1	32.146s	24.108s
GFfunc	4660	7.558s	7.558s

Table 3.5: MATLAB profile of the simulation with 5000 time points.

With these statistics, scaling the performance as with the original simulation to a million runs would involve timings considerably longer, as depicted in Figure 3.3.

With this information, we can see that if one run takes approximately 30 seconds, then 1000 can complete in under a day and 100,000 would take approximately 5 weeks. The initial 127 day time predicted for the million runs has also slowed down to take over a year, making this work at this accuracy of 5000 time points per run even more computationally infeasible. Nevertheless, we continued with the ambition of maintaining our previous target of a million runs with this accuracy in under one day. Our under 3 hour implementation uses this accuracy, and therefore demonstrates a 2976 times faster runtime.

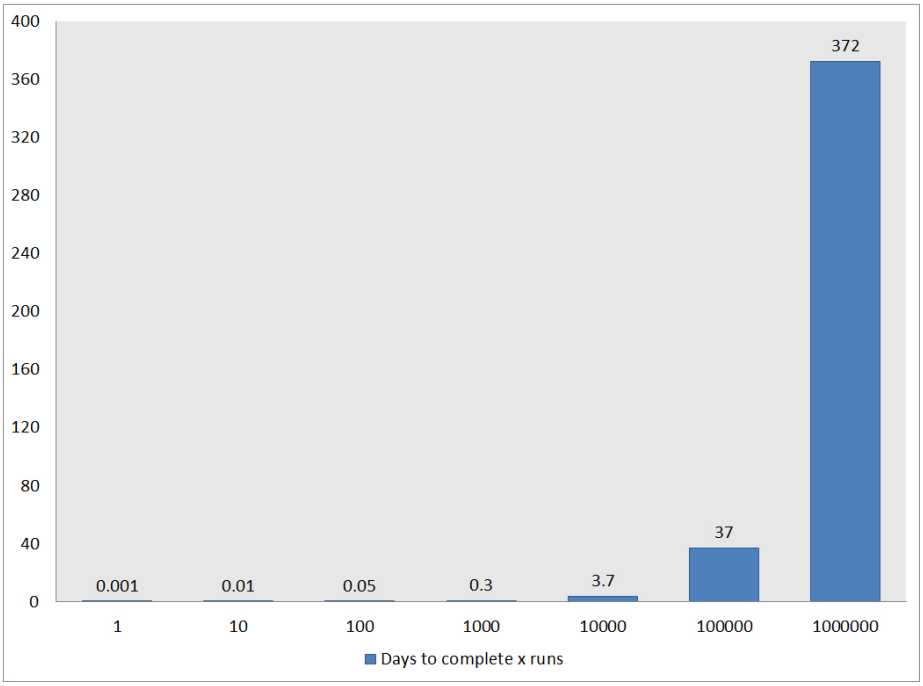

Figure 3.3: Amount of days required to complete various runs of the new simulation.

3.4.2 Custom Types

The Coder generates C++ files which directly represent the MATLAB functions translated. For the primitive MATLAB types, however, custom classes are generated. These represent elements such as matrix variables. The types are defined in two additional header files that provide the following trivial replacements:

```
typedef int int32_T;
typedef unsigned int uint32_T;
typedef double real_T;
typedef unsigned char boolean_T;

typedef struct emxArray_real_T {
    real_T *data;
    int32_T *size;
    int32_T allocatedSize;
    int32_T numDimensions;
    boolean_T canFreeData;
} emxArray_real_T;
```

Coder assumes compilation on a target using a twos complement representation for signed integer values. Structs are used to represent matrix variables for speed during lookup and manipulation. In structs, members are stored in memory in a contiguous fashion, therefore accesses to data can be significantly faster than an equivalent class or object representing a matrix variable stored on the heap. Since the struct holds information regarding the allocated size of the matrix and its dimensions, this information need not be inferred at runtime and can be used explicitly to make assertions about the variables, assuming they have not been modified by code other than the built in emxArray functions.

3.4.3 Pseudo-Random Number Generator

The simulation utilises a pseudo-random number generator (PNRG) in order to obtain values to update the time steps within the Kinetic Monte Carlo model. MATLAB specified this as a generator which provided uniformly distributed numbers. The implementation of this when translated to C++ was a Mersenne Twister.

Developed in 1997 by M. Matsumoto and T. Nishimura[19], the Mersenne Twister uses a period length that is a Mersenne Prime, described earlier in the background as a prime number of the form $2^p - 1$. The translated simulation uses the MT19937 implementation with a 32-bit word length. This ensures that the random numbers are almost uniformly distributed within the range $[0, 2^k - 1]$.

This gives the algorithm a long period length of $2^{19937} - 1$, desirable for use in long term simulations. Shorter period lengths may result in more frequent duplicate results, and most other implementations use lengths of 2^{32}. Therefore, it was in our best interests not to modify the choice of algorithm. MT19937 also passes various tests for statistical randomness, making it an ideal choice for our simulation and therefore allowing it to closely represent random transitions between biological XB states.

The algorithm has the following method signature.

```
static void twister_state_vector(uint32_T mt[625], real_T seed)
```

It is called using b_rand(real_T) by providing a double-type seed from which to begin generating values. This seed was initially set to 5489.0 by the Coder after translation. Using a fixed seed has the problem of causing the PNRG to give the same sequence of values on every set of calls. The consequence of this was that every run of the KMC simulation in C++ would return the same Force/Time pairs. 1,000,000 runs with the same result would not be representative of the muscle contraction system, so we needed to ensure each call to b_rand used a different value as its seed. We fixed this by including the ⟨time.h⟩ C++ library and using (double)(time(0)) as the seed.

Since a call to time(0) returns the current POSIX time and is cast to a double before use as a seed, the value will be different on each run of the simulation, it will take time to complete one run and move on to the next.

POSIX time is defined as the number of seconds that have passed since January 1 1970, with prior times negatively defined. For instance, the date at which A. F. Huxley's derivation of the Sliding Filament Theory was published (22 May 1954) is represented as -492696000 (using midday).

We were able to leverage the biased results given by the fixed seed of 5489.0 to test the correctness of the C++ translation. This was done by altering the MATLAB function to use only that fixed seed, causing it to provide biased results. The following code was added to the KMCmanyfunc.m.

```
rng(5489.0, 'twister');
```

We then compared the output from the MATLAB to the C++, and if the Force/Time pairs were exactly the same, we could assert the correctness of the C++ and any future changes. We built in a debug mode into the C++ that would run the simulation using the fixed seed and automatically compare the outputs to the original MATLAB, by way of a test wrapper. At this point, the simulation in C++ was shown to be correct in representing the MATLAB exactly.

3.5 Optimisation Gains

Performing profiling analysis on the C++ version using the GNU Profiler for C and C++, gprof, demonstrated the timing of one run of the KMC simulation to take approximately 5.43 seconds. This is a significant reduction considering the C++ contained various inefficiencies that were a result of modifying the MATLAB to allow Coder to automatically translate. These included running the LUT generation code on every run of the simulation, slowing down the process greatly.

Function Name	Total Time	% of time
eml_lusolve(emxArray_real_T const*...)	4.14s	86.0%
GFfunc(double const*, double const*...)	0.24s	14.2%
KMCmanyfunc(double*, double*)	0.05s	10.2%

Table 3.6: C++ profile of the simulation with repeated LUT generation.

Table 3.6 shows some of the most time consuming method calls within the C++ simulation. Having spent over 4 seconds in the eml_lusolve function, the next step is to analyse its purpose and perform the necessary optimisations directly on the C++ code to minimize its execution time.

3.6 Summary

In this chapter we have demonstrated a speedup of over 50% by departing from the MATLAB language and environment. By using the Simulink Coder to

translate the KMC simulation into C++, the program runs in a shorter time period although having done more work. Some inefficiencies were added in order to be able to make the automated transition to C++ possible, such as the inclusion of the Look Up Table (LUT) generation code and full implementations of relevant MATLAB built-in functions which were present.

We show that the compiled language of C++ is inherently faster, but to reach this stage it was demonstrated that the script first needed to be transformed into a function. Functions are considered generally faster than scripts, performance wise. Since the Coder required all variables to be declared before use, any matrix variables were preallocated, preventing the need for the compiler to infer sizes and allowing faster processing. The use of the Mersenne Twister algorithm for random number generation provides a uniform robustness with its generated values over a large amount of runs, making it the ideal for our target of a million runs.

Writing a wrapper around the C++ allowed easy testing and profiling of the simulation, including the ability to test whether any code changes have resulted in breaking the correctness of the underlying simulation. We seed the MATLAB and the C++ program with a fixed value to compare the results, asserting correctness if the outputs from both are exactly the same.

Chapter 4

Optimising the KMC Simulation in C++

In this chapter, we present an optimised version of the KMC simulation code in C++ to minimise the execution time and allow performance gains which make the target of one million runs feasible. The C++ code contains built-in translated MATLAB functions that are only used for a fraction of their intended purpose. We can optimise the code further by eliminating the generic nature of these functions and rewriting them to only perform the specific and smaller task that we require.

Furthermore, we were required to add code for LUT generation within the simulation itself for the automated translation to be possible. This causes every run to rebuild the 8x20001 matrix of known values, and is a massive contributor to the program's slowdown. We use the following loop that wraps around our call to the simulation to allow differentiation between running once in our test debug mode with a fixed PNRG seed or many times via a command line parameter.

We present to the reader in this chapter a new C++ model complying with the following optimisations.

- A refactoring of methods to utilise less resources.
- A reduction of calls to heavy computation such as LUT generation.
- Redefinition of functions to increase code flexibility and versatility.
- Profiling information depicting the performance gains after each optimisation.

We verify the correctness of the changes made by fixing the random number generator seed as before and comparing the simulation output values to the original unchanged MATLAB. This will ensure that the semantics of the simulation have not been modified. If the values are not identical, however, we can assume the simulation is no longer correct.

4.1 Refactoring

Refactoring is the process by which an internal representation of code is changed to improve efficiency, structure or readability without changing the program's external behaviour. The advantage of refactoring code is that its complexity will be reduced, allowing it to be easy to maintain and modify in future, as well as making optimisations easier to recognise. Its usefulness ranges from rapid development to hardware implementations[20]. Since there are various means by which to perform a task in code, we identify whether the means used throughout the simulation, including that generated by the Simulink Coder, are the best possible to perform our tasks. We show that the Coder used heavy implementations of complex mathematical functions in order to solve simpler problems, and use an approach to overcome this issue without disrupting the underlying layers.

Some elements to identify for refactoring include the following, which may be reminiscent of larger problems within the code.

- Code which is duplicated over various methods.
- Methods which have outgrown their intended functionality.
- The use of complicated design patterns to solve simpler problems.
- Large parametrization of functions and methods which hinder readability.
- Excessively short variable names which hinder maintenance.

We consider pseudocode and its benefits in inspecting the desired algorithm of parts of our simulation and MATLAB translation in order to be able to modify the code on an implementation independent basis. This allows us to make changes which are less likely to disrupt the control flow of the simulation, as well as preserving variables which are utilised throughout.

The first inefficiency we identified involved the Look Up Table (LUT) being generated on every iteration of the main loop wrapper, causing it to call on each run of the simulation.

```
for (int i = 0; i < nRuns; ++i) {
  ...
  KMCmanyfunc(times, outputs);
  ...
  if (debug_mode) {
    boolean_T matched = check_values_match(times, outputs);
    printf('Results %s', matched ? 'PASS' : 'FAIL');
    }
}
```

We optimise the KMCmanyfunc code to only run the LUT generation code by adding an extra parameter into the call to KMCmanyfunc. By passing in the loop variable i, we can keep track of progress and the current run.

```
KMCmanyfunc(times, outputs, i);
```

We then modify the method signature of KMCmanyfunc to include the following currentRun variable. This will give the simulation the ability to be aware of its own progress in the batch of runs.

```
void KMCmanyfunc(real_T OutputTime..., int currentRun)
```

By making sure the LUT generation code only occurs on the first run, we save execution time by not running the same code many times unnecessarily.

```
if (currentRun < 2) {
  // Create LUT only if first run
  ...
}
```

Initially, this was a problematic addition to the code, since the scope of some variables which were reused later on in the simulation had been changed to be visible only in this selection statement. The Simulink Coder performed optimisations when automatically translating from MATLAB by reusing declared matrix structs elsewhere. The problems which arose were that during the selection statement in the first run, some variables such as the emxArray_int32_T, declared as ii, were sent to an initialisation function such as emxInit_int32_T. During the first run of the simulation, these initialisation functions would correctly be called, but during any future runs the program would report a segmentation fault due to bad memory accesses. This was the case because the initialisation functions were never called. To rectify this, we isolated the arrays which were reused, and provided conditional initialisation for them on runs greater than the first.

```
// initialise matrices if LUT already exists
if (currentRun > 1) {
  emxInit_int32_T(&ii, 2);
  emxInit_real_T(&Q, 2);
}
```

After performing these corrections we were able to assert the correctness of the program by performing many runs in debug mode to confirm a sustained 100% match with the original MATLAB using the fixed PRNG seed.

4.2 Profiling

Ensuring the LUT was generated once only was a big step in reducing the amount of processing required for each run. From this stage we can begin to optimise the remaining functions spending the most time as reported by the gprof profiler tools. By running the code more than once it becomes apparent

Function Name	Total Time	% of time
eml_lusolve(emxArray_real_T const*...)	1.38s	78.86%
eye(double, emxArray_real_T*)	1.60s	12.57%
GFfunc(double const*, double const*...)	1.64s	2.29%

Table 4.1: C++ profile of the simulation with once-only LUT generation.

that there has been a significant speedup. The profiler reported this version finishing two runs in 1.75 seconds.

Due to the portability of C++, we were able to import the code into Visual Studio and attach a debugger for analysis with breakpoints. This led us to realise the reasons why segmentation faults were occurring in less time and understand what needed to be done in order to rectify the problems. Table 4.1 highlights the remaining slow functions and their child processes. We now take a close look at these functions and discuss steps which were taken to optimise them.

4.3 Reimplementing MATLAB Functions

We observe that the eml_lusolve and eye function calls take the most time, with a significant gap between their execution lengths. The eml_lusolve function corresponds to an LU decomposition. In MATLAB, our GFfunc.m function performed a matrix inversion during each function call. Coder was unable to determine the semantics of our matrix inversion and therefore translated the code into C++ to be handled by a more generic function that could perform a wider range of tasks.

LU decomposition is the factorisation of a matrix to form the product of a lower (L) and upper (U) triangular matrix. It is analogous to Gaussian elimination and is fundamental in solving systems of linear equations, performing an inversion or calculating determinants. Introduced by A. Turing in 1948, the tool understandably powers many of MATLAB's underlying matrix manipulation functions, trading efficiency for the ability to solve a range of problems.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} . \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Figure 4.1: An LU decomposition of a 3x3 matrix.

Figure 4.1 depicts an LU decomposition of a 3x3 matrix into an LU product. Since this is the most complicated and inefficient part of the code, we consider this for acceleration in chapter 5. It has become apparent that the Simulink Coder habitually implements small and special case operations from MATLAB to be solved under heavy one-size-fits-all operations in C++ such as the LU

solver for a matrix inversion. Further evidence of this is gathered by examining the eye function and how Coder has changed the intentions of our code.

In the GFfunc.m MATLAB code, we see one instance of this function call: $Eq = eye(V)$. Our intention was to generate a $V * V$ identity matrix for later use. The MATLAB eye function, however, is overloaded to have various applications. The MathWorks documentation describes the following possible calls to eye.

```
Y = eye(n)
Y = eye(m,n)
Y = eye([m n])
Y = eye(size(A))
Y = eye
Y = eye(m, n, classname)
```

The Simulink Coder implemented eye with all of its possible forms due to our single use. Since the C++ implementation of eye includes hard-coded values which are seemingly indecipherable, it would have been an arduous task to redefine the translated version and test its correctness afterwards. A workaround that guarantees correctness is to use the Simulink Coder as a black box to ensure our code is translated while being semantically correct. We wrote our own MATLAB function to perform our intended calculations using eye and used the Coder to translate this function to C++.

```
function [A] = newEye (x)  %#codegen
A = zeros(x);
for i=1:x
   A(i,i) = 1;
end
end
```

When translated, this function in C++ was only 14 lines of code (compared to the original eye's 29) and the method signature remained the same.

```
void eye(real_T n, emxArray_real_T *I)
```

We replaced eye in the C++ with this new version and a debug run confirmed the correctness of the code. We looked for various built in functions that we could optimise in this manner and profiled the resultant program to obtain our final C++ implementation.

This approach of using the Coder as a black box was paramount in modifying the C++ code without making significant errors or disrupting variables used elsewhere. Another issue with Coder was that the translated program was heavily optimised in the way it used variables, so we had to be careful when removing seemingly unused values.

4.4 Functionality & Performance Gains

To prepare the simulation to run our long term experiments, we would need to not only be able to set the number of runs, but also specify the actin compliance (Ca), myosin compliance (Cm) and the compliance of the crossbridges (Cx). These values were currently hard-coded in the simulation as $Ca : 0.0002$, $Cm : 0.0021$ and $Cx : 0.5$. We updated the C++ program to accept an increased amount of command line (CLI) arguments than just the number of runs to be made, and provided a mechanism to revert to the hard-coded values in case some of the CLI input was non-existent.

```
if (argc > 1) {
  nRuns = atoi(argv[1]);
  Ca = (argc > 2) ? atof(argv[2]) : 0.0002;
  Cm = (argc > 3) ? atof(argv[3]) : 0.0021;
  Cx = (argc > 4) ? atof(argv[4]) : 0.5;
}
```

The parameters in the method signature of the KMCmanyfunc function was extended to reflect these new parameters.

```
void KMCmanyfunc(...real_T Ca, real_T Cm, real_T Cx...)
```

The frequent use of bitwise operators to perform operations in a nontrivial manner provided very strong optimisations to the code. Coder was able to comfortably identify cases where the compiler's optimisations would not disrupt our correctness, thus trading readability for a reduced execution time in a manner of compromise.

```
for (ix = 0; ix < 20001; ix++) {
  LUT[ix << 3] = R1[ix] * Ratio1[ix];
  LUT[1 + (ix << 3)] = GA2[ix] * Ratio2[ix];
  LUT[2 + (ix << 3)] = R3[ix] * Ratio3[ix];
  LUT[3 + (ix << 3)] = GA3[ix] * Ratio4[ix];
  LUT[4 + (ix << 3)] = R1[ix];
  LUT[5 + (ix << 3)] = GA2[ix];
  LUT[6 + (ix << 3)] = R3[ix];
  LUT[7 + (ix << 3)] = GA3[ix];
}
```

This extract from the LUT generation code demonstrates the use of shifting to cover all assignments in the 8x20001 matrix struct in one loop. Debug runs using our test wrapper on the fixed seed comparison with the MATLAB code confirmed all our changes up to this point as correct, and the gprof profiler reported this iteration of the C++ implementation as taking 0.93 seconds for one run.

Table 4.2 shows a clear decline in the time spent within eye, along with the decline in running times for the overall KMC simulation.

Function Name	Total Time	% of time
eml_lusolve(emxArray_real_T const*...)	0.64s	68.82%
eye(double, emxArray_real_T*)	0.19s	20.43%
GFfunc(double const*, double const*...)	0.04s	4.3%

Table 4.2: C++ profile of the simulation after final optimisations.

4.5 Experimental Considerations

At this point, it was important to review the steps we were to take in performing our experiment to find out more about the Force-Velocity relationship on the muscle crossbridges (XBs). Since our KMC simulation does not treat the XBs as independent, any long term results we obtain from our runs will be the first measured of this kind. It was therefore important to take advantage of the simulation's flexibility and obtain as much meaningful data as possible for later analysis.

The steps for moving on from this point and obtaining experimentally viable data were outlined as follows:

- Run the simulation 1,000,000 times with default compliance values Ca : 0.0002, Cm : 0.0021.
- Analyse the results with a moving average and obtain a grand list of the Force deltas.
- Plot the Force-Time pairs to examine the relationship.
- Scale down to fewer runs, n, while maintaining a reasonable amount of smoothness in the graph.
- Run the simulation n times with new compliance values 10x smaller Ca : 0.00002, Cm : 0.00021.
- Run the simulation n times with new compliance values 2x larger Ca : 0.0004, Cm : 0.0042.
- Analyse the latest runs to obtain grand lists of Force deltas and create plots.
- Plot the three graphs atop each other.

Since the XB compliance is well known and not known to fluctuate on the same scale as the actin and myosin compliances, we do not vary it from its value at Cx : 0.5. We effectively reject the null hypothesis if there is a difference between the three graphs when plotted in an overlaid fashion. This is due to any differences in the graphs suggesting that the life of an XB is affected by the compliance of its environment. This has never been measured and can lead to publishable data. If the graphs are identical, however, then we cannot make any such inference. We discuss our analysis in chapter 6.

4.6 Summary

In this chapter we have demonstrated an array of optimisation methods in the C++ simulation, including rigorous use of bitwise operations, variable reuse as directed by Coder and conditional resource allocation to ensure the fewest resources necessary are utilised. We explore the non-standard notion of using the Coder's automated translation feature as a black box to interpret our semantic meaning while coding and to provide C++ implementations directly mirroring this meaning. We see such optimisations in functions as eye, which was necessary due to the Coder naively translating built-in MATLAB functions to larger and more inefficient C++ counterparts.

We present the slowest part of the code as the matrix inversion having been translated to an LU decomposition function, and argue that acceleration of this aspect is the key in running the KMC simulation 1,000,000 times in a realistic timeframe. We show that the C++ program has been further optimised to run a simulation in under 1 second, and add functionality to the wrapper to allow various changes. We explain that this was made possible by refactoring the code and making optimisations that we were unable to otherwise perform in the MATLAB environment. This, and the speedup gained, justifies our decision to migrate from MATLAB into C++.

With the ability to change the actin and myosin compliance values between simulation runs, and the ability to define the number of runs which should occur all from the command line, we begin to focus on the experimental implications and possible outcomes of our data. We therefore outline the steps required to be able to reach a point where further research is necessary to better understand the results of our data.

Chapter 5

Accelerating the KMC Simulation

In this chapter we present our accelerated solution to running the KMC simulation and inform the reader of the choices made along the process. Since we identified the most time consuming part of the simulation fundamentally as a matrix inversion problem disguised as an LU matrix decomposition, and since we have optimised the C++ simulation to what we believe was the maximum possible extent in the timeframe of this project, we look to acceleration to make our target of a million runs feasible.

5.1 Resource Considerations

Our primary concern with acceleration is the volume of data which will be generated as output. Since we run the program 1,000,000 times with each run performing a KMC simulation of actomyosin interaction during muscle contraction, we collect up to a maximum quantity of 5000 Force/Time pairs. This is due to the simulation of the system in random time steps up to 0.1s. This allows us to make forward projections from estimating one run's output as 70KB, ten runs as 700KB and the target million outputting approximately 70GB of data.

This poses the problem of I/O bottlenecks, and we must take these into consideration whether running locally, on the FPGA hardware, on GPUs, or on Condor. Running the current optimised C++ simulation sequentially on a local machine to perform 1,000,000 runs will not allow us to reach the target runtime of under one day.

Figure 5.1 shows the runtime of our optimised simulation generally taking an order of magnitude less to run. Since one run completes in under a second (0.94s), we can extrapolate to determine that 10,000 runs will take approximately 2.5 hours, with 100,000 taking just over a day. The target million runs executed in this manner will take 10 days. This is a significant speedup from the previous runtime which took many months to complete, but is still not feasible.

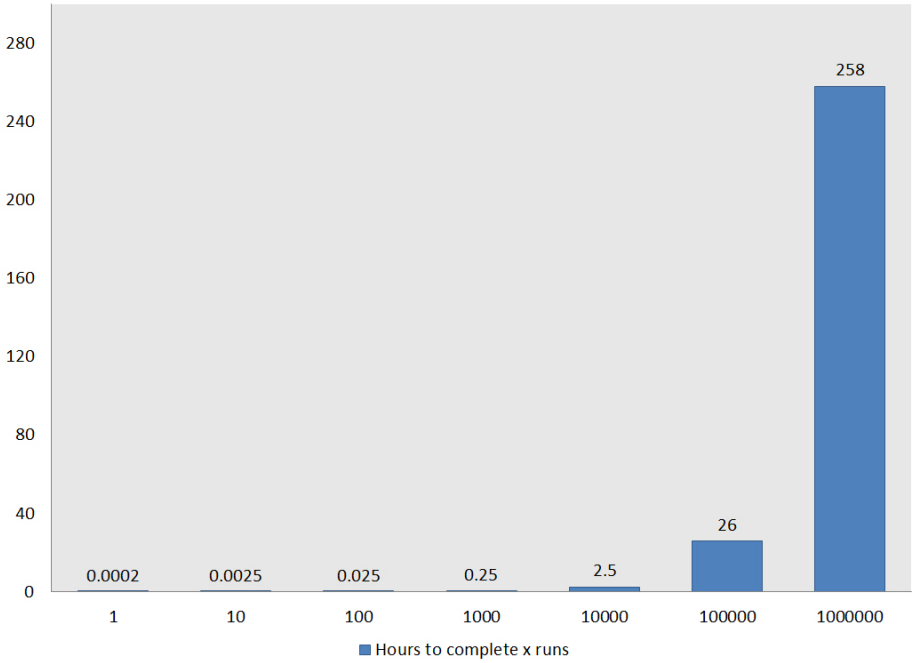

Figure 5.1: Amount of hours required to complete various runs of the optimised C++ simulation.

We therefore identify the need for parallelism and rule out the possibility of running sequentially.

5.2 FPGA Acceleration

While FPGAs would provide massive parallelism by way of their numerous dataflow cores, the translation to VHDL or MaxJ was deemed an arduous task which would be extremely time consuming when considering the additional problems to be overcome. Studies have demonstrated implementations of LU decomposition algorithms on FPGA hardware[21] and GPUs[22], though the availability of Condor and its ability to parallelise the C++ simulation over many machines, CPUs and cores without modification made it the most optimal choice. We therefore leave both FPGA and GPGPU implementations of the simulation as potential extensions if necessary.

In addition, since the FPGA dataflow engine operates as a slave under the control of a CPU, we would experience a bottleneck at the connection over which we transfer the inputs and receive the outputs. This is the PCI express port (PCIe). While some studies have demonstrated the use of data compression over PCIe[23] on FPGAs, we believed this, coupled with the inevitability of

transferring 70GB of output data, would add complexity to the algorithm and create a slowdown that may significantly offset the speed of processing.

Compilation for simulating an FPGA run ranges from at least 30 seconds to several minutes, slowing down the debug process greatly, and compilation for hardware runs ranges from at least 20 minutes to potentially several days. While FPGAs would speed up the matrix inversion, they would need to wait for the inputs to be refreshed, and the additional latency over PCIe would become apparent over a million runs. These disadvantages prevent the FPGA hardware from being our first resort to acceleration.

5.3 High Throughput Acceleration

Condor treats the connected network of machines as a graph in which every machine's available CPU core has been configured as a 'slot' on which jobs can be scheduled to run. When Condor receives a job, it distributes the executable across many slots (and therefore machines) to parallelise on not only between machines, but also their CPUs and cores. The jobs are scheduled to run in a cycle stealing mode and we deploy our C++ simulation on this system to take advantage of the 4-CPU dual core setup. With each dual core CPU clocked at 3.2GHz, we see 8 logical cores per machine, and with over 200 machines we see a significant amount of parallelism over strong modern hardware.

Condor jobs are submitted by way of a command script which describes the executable file to be run and the amount of times (jobs) to perform. This script can also specify the command line arguments which are needed, useful for our simulation. All output written to stdout or stderr by the executable is saved as a unique file corresponding to the job number in a shared space that all Condor nodes can access. We therefore submit our jobs from a shared network area and write all output pairs to stdout.

We present to the reader in this section a solution which takes advantage of the Condor High Throughput Computing system by distributing many requests to run a simulation (jobs) across various machines in a grid-like manner. By running the executable on many machines, we expect to invoke significant levels of parallelism and to run not only across machines, but also between CPUs and cores on a single machine. This will allow us to complete the long term simulation runs vastly quicker than a sequential run on one machine. This method of parallelism is possible because the simulation is self contained and does not depend on a previous or future run in order to produce results.

We use the following command script to submit our jobs. From this it can be observed that we have greater flexibility over the granularity of the runs, and we initially submit 1,000,000 jobs to run 1 simulation each.

```
universe = vanilla
executable = KMCmain
output = kmcmain.$(Process).out
error = kmcmain.$(Process).err
```

```
log = kmcmainlog.log
arguments = 1
queue 1000000
```

The output files are named according to the process that was dispatched to execute the job, which would result in 1,000,000 unique output files of 1 run each. This method was shown to be problematic, since submitting 1,000,000 jobs took a long time, and keeping track of progress was a cumbersome task. This would result in 2,000,001 output files (1,000,000 outputs and error logs with one Condor log). We therefore examined the optimal grain size for our target runs.

We had various options regarding granularity, and were mindful of the potential for compression of data when submitting many scripts rather than fewer jobs with more simulation runs.

Condor Scripts	Jobs Per Script	Runs Per Job
1	1,000,000	1
10	10,000	10
10	100,000	1

Table 5.1: Granularity options for running the simulation on Condor.

To balance and be able to add compression between scripts, we elected to submit 10 scripts to run 10,000 jobs each with each job running the simulation 10 times. This resulted in 10 batches of 10,000 output files each containing 10 runs of the simulation in a machine readable format. This decision also took advantage of the Condor scheduler's behaviour, in that it would give priority to jobs which had a higher run time. By setting the KMC simulation wrapper to perform 10 runs, each job would take enough time not to be held in the queue to make way for other jobs.

5.3.1 Correctness

After obtaining our million output results in the form of 100,000 files containing 10 runs each, we noticed two interesting aspects of the data. The first was that some output datasets of Time/Force pairs were identical, suggesting a problem with the algorithm or method of acceleration. Since we had proven the C++ algorithm to be correct by way of comparison to the original with a fixed seed, we looked to the random number generator to explain this behaviour.

The seed for the random number generator (RNG) was the current POSIX time, and Condor was parallelising many jobs to run across different machines at the same time. This meant that although the RNG seed would be different between the ten runs per job due to the simulation taking time to complete, it was possible and very likely that various jobs running at the same time across different machines would begin with the same seed as they were scheduled by Condor and the system time was derived from a central server on the network.

This resulted in many runs having duplicate data, and suggested we only had a subset of the unique 1,000,000 runs we aimed to obtain.

Condor allows referencing the job number and host name of the machine, but since a machine's CPU cores are considered nodes, the host name would not be unique. Our only uniquely changing information was the POSIX time and job ID. Furthermore, it was feasible that since the job number reset every 10,000 (10 scripts containing 10,000 jobs each were submitted), we may have duplicate seeds if we continued to use the time. This would occur due to the possibility that some high job ID (9999) added to an early time may match a future time added to a low job ID (1), thus matching the equivalence: $t + 9999 \equiv \Delta t + 1$. To be able to generate 1,000,000 unique seeds for the RNG, we needed to be certain that no collisions were possible, so we sought to rethink our approach to the problem.

We updated the method signature of KMCmanyfunc within the C++ to accept the Condor job ID as a parameter, and calculated our seed based on this value. The job ID was passed in to the C++ program as a command line input, and one script for 100,000 jobs was submitted instead of the previous arrangement. This ensured the generation of 100,000 files with 10 simulation runs each, and allowed us to be certain of the job ID uniqueness.

```
void KMCmanyfunc(int jobID, ..., int currentRun)
```

By keeping track of which job the simulation was running under, we were able to use this information coupled with the *currentRun* variable to ensure that 1,000,000 unique seeds were used across all the runs. We replaced the seed of (double)(time(0)) with the new value calculated by: $(10 * JobID) + currentRun$, where *currentRun* ranged between 0 to 9 depending on which simulation out of ten was running in a job. This resulted in the following behaviour, giving ten unique values per job and preventing collisions across jobs.

Job ID	*currentRun*	Seed
0	0	0
0	1	1
0	2	2
0	...	...
0	9	9
1	0	10
1	1	11
...	...	...
99999	0	999990
99999	1	999991
99999	...	...
99999	9	999999
m	n	$10m + n$

Table 5.2: Random number generator seed uniqueness between runs.

Table 5.2 demonstrates that our new seed calculation will provide unique values across all 1,000,000 runs of the simulation provided that they are submitted as one batch of 100,000 jobs in a script. As a result, the job ID would need to be passed in as a parameter, so we updated the command script to submit jobs to reflect this change.

```
universe = vanilla
executable = KMCmain
output = kmcmain.$(Process).out
error = kmcmain.$(Process).err
log = kmcmainlog.log
arguments = 10 $(Process)
queue 100000
```

We therefore run the simulation using this configuration of 1 command script with 100,000 jobs performing 10 runs each to obtain our 1,000,000 correct results and observe the run times of Condor.

5.4 Profiling

The Department of Computing's Condor system at Imperial College has an average of approximately 100 jobs being active in a sustained manner throughout the year. Despite being a High Throughput System, it is occasionally used for tasks which would otherwise be suited to High Performance Computing.

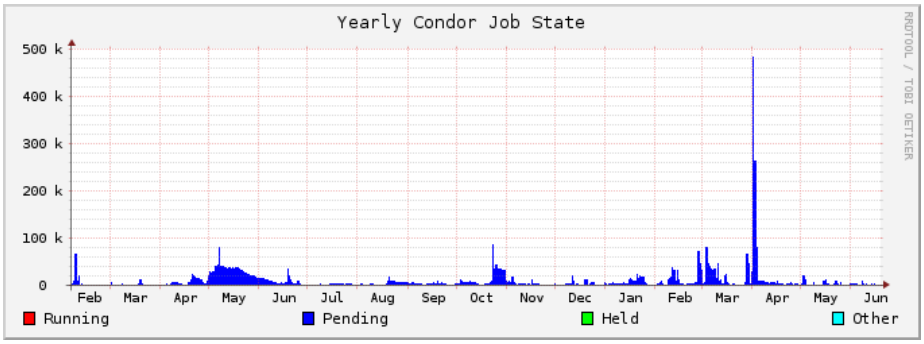

Figure 5.2: Condor job queue state over a period of one year.

This can be seen in Figure 5.2, where small spikes occur to denote the submission of large jobs. The Condor pool's utilisation statistics do not generally exceed 20,000 jobs, reaching an upper limit of 100,000 in rare cases. Our submission of the initial 1,000,000 jobs is clearly indicated on the graph as a spike up to 500,000 jobs (the limit for the drawing software was reached). The amount of jobs currently in the Condor queue, coupled with the job submission time, affect the throughput of the system in general. If our jobs were submitted at a

time when the machines were subjected to physical use, those users would have priority over the Condor system. This is a consequence of the cycle stealing mode under which it runs.

We were able to run the 100,000 jobs within a timespan of under three hours. This provided us with 1,000,000 results at a rate of approximately 93 per second, translating to an approximate throughput of 10 jobs per second. This value varied, however, due to the state of the machines, and we attempted to submit our jobs at the most convenient times for the least physical use. The size of the data was, close to predictions, 68.8GB.

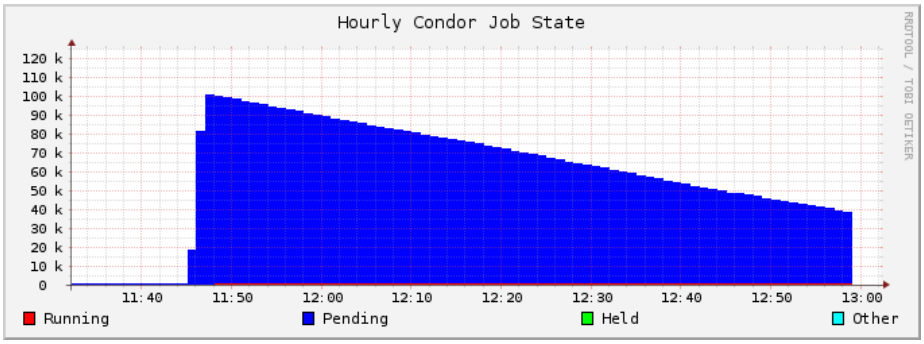

Figure 5.3: Condor state for 100,000 jobs running the KMC simulation.

Figure 5.3 shows the progress of the 100,000 runs of the simulation completing in under three hours. Having identified that 1,000,000 Force/Time pairs produced a result set far more sufficient than necessary, we were able to scale down the number of runs required in order to obtain experimentally viable results to save time and other resources. We tested with various values and re-ran the simulation to obtain 250,000 results by submitting only 25,000 jobs with the new compliance values which were ten times smaller (Ca : 0.00002, Cm : 0.00021) and two times larger (Ca : 0.0004, Cm : 0.0042).

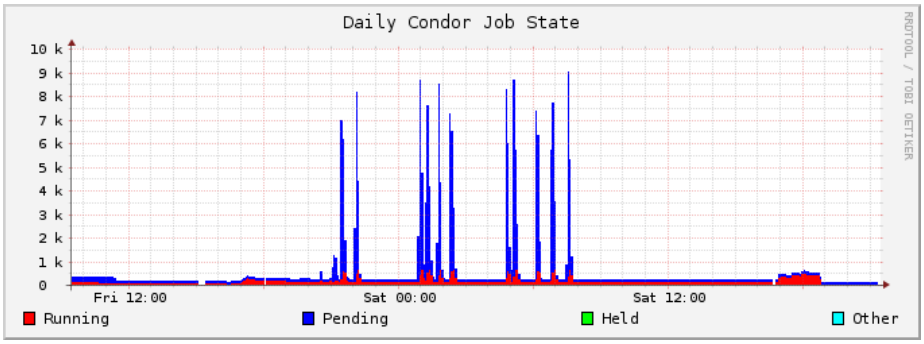

Figure 5.4: Condor job queue state over a period of one day.

Running 25,000 jobs only took approximately 30 minutes, giving a throughput of approximately 14 per second. Since there were fewer runs, the throughput was lower as the Condor scheduler gave more priority to other jobs. Figure 5.4 shows the daily activity regarding the Condor queue (in blue) and the running jobs (in red) during the day. A maximum of 500 jobs run simultaneously to provide parallelism, with others held until enough are completed.

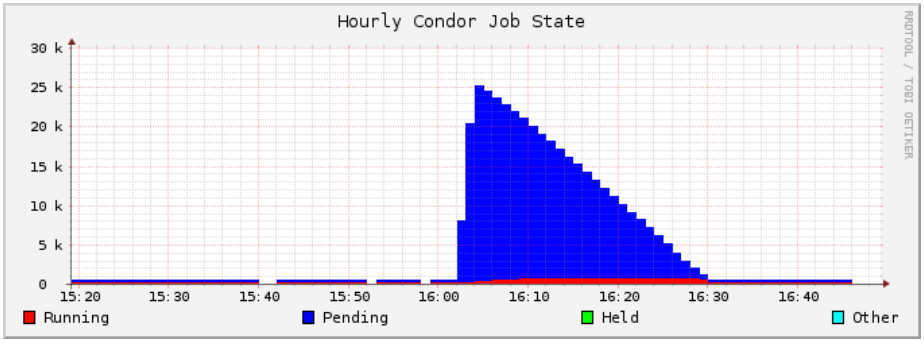

Figure 5.5: Condor state for 25,000 jobs running the KMC simulation.

Figure 5.5 shows the Condor queue status during the 25,000 job submission (250,000 runs). With this method of acceleration, we were comfortably able to obtain our results within a reasonable timeframe and were able to continue onto analysing the Force-Time outputs of these biological systems and compare the crossbridges under different compliant environments.

5.5 Summary

In this chapter we have demonstrated the shortcomings of FPGA acceleration with regard to our project and have shown that the Condor system was sufficient in parallelising the simulation to obtain the target million results comfortably within three hours. We show that the machine nodes present in Condor were busier at different times and take advantage of this by submitting our jobs strategically.

Our target output initially resulted in duplicate results, rendering the perceived million only a subset. We rectified this by ensuring the random number generator was seeded with unique values covering the range of the number of simulation runs expected. Once we obtained our target million runs, we performed further simulations to obtain outputs for two sets of different compliance values, representing a higher and lower compliant environment. These jobs, corresponding to 250,000 runs, completed in 30 minutes.

We were able to overcome the I/O bottleneck problem by utilising a shared space which all Condor nodes could access, with each machine writing to one directory. This ensured the output files were all saved in a central location under a unique name. After obtaining the result sets for low, normal and high

compliance values, we prepared to perform experimental analysis and compare these never before seen measurements to infer conclusions.

Chapter 6

Experimental Analysis

In this chapter we present novel results which suggest the lifespan of crossbridges are affected by the compliance of their environment. We present the reader with methods of analysing the data from our Kinetic Monte Carlo simulation and plotting the results in a graph, detailing the issues which arose as a result of handling approximately 70GB of numerical data.

We infer from these measurements, the first of their kind, that the lifetime of a crossbridge is less in a more compliant environment, and discuss the implications of this research along with the possibility and benefits of further investigation into why this is the case.

6.1 Voluminous File Handling

Since we were in possession of outputs from over 1,000,000 runs and since each of these runs contained up to 5000 Force/Time pairs, we were required to deal with both the numeric quantity of files as well as the volume of data contained within. This data amounted to over 70GB of results that we desired plots from.

We were first required to separate the three batches of m files each containing 10 runs into n files containing 1 run each. We wrote an algorithm for this using C# due to its robustness, debug capabilities and effectiveness in rapid development.

```
for (int i = 0; i < 100000; ++j) {
  string input = readFile( "output." + i + ".out";)
  string[] tenRuns = Regex.Split(input, "\n\n");

  for (int k = 0; k < 10; ++k) {
      file.Write(tenRuns[k], "/Expanded/output");
  }
}
```

This code read each of the files Condor generated in one batch script, expanding them by locating the separation character and creating new files representing one run each. The reason this was necessary was due to the limitations in the string handling and data loading methods within MATLAB. We were able to confirm using this program that each of the files Condor output included ten runs.

After the results were expanded, analysis was undertaken by an algorithm which would perform a moving average of the three datasets representing low, medium and high compliant environments representative in the stiffness of the actin and myosin filaments. The algorithm involved undertaking the following steps.

- Read in all relevant files containing simulation runs.
- Iterate through list to separate times and forces into two grand lists.
- Calculate list of force differences for each time ΔF.
- Generate new grand list of time t and force difference ΔF.
- Sort grand list by time.
- Determine average of forces over time.
- Plot time against force to obtain final graph.

The problems with this version of the averaging code revolved around I/O bottlenecks due to the voluminousness of the input. Since there were 70GB of files, reading them all in to produce an initial grand list required 70GB of RAM, a specification not available on most typical machines. We attempted to extend our page table size by allocating sectors of our machines' hard disks to be used as virtual RAM, but the inclusion of steps which required more grand lists to be generated would cause out of memory errors very quickly. This forced us to rethink the algorithm.

The speed of the underlying memory became a bottleneck at this point, since it was dependent on the relative speed of the file handling I/O mechanism. The hierarchy of I/O latency ranges from fastest to slowest in the following manner, depicted by the proximity to the CPU and internal speed of the device.

- CPU L1 cache.
- CPU L2 cache.
- CPU L3 cache (if present).
- Main memory (RAM).
- Secondary memory (Hard Disk).

Since we were reading from the hard disk with a machine that had only 4GB of RAM, the page table would fill up very quickly causing many page faults. At this point the hard disk would need to be consulted to fill in the missing information, but restrictions would be imposed on MATLAB for the maximum possible array (and total space available for arrays) through the page table size (combining RAM and page file size).

We therefore rewrote the averaging algorithm to eliminate the need to create a grand list beyond the maximum allowed size using interpolation on the input data.

6.2 Interpolation

Interpolation is a mathematical device used when large datasets are handled in order to obtain new data points within the known range. These methods, including regression analysis and curve fitting, allow the approximation of functions from their graphs. By performing linear interpolation, we can reduce the need to create a grand list which exceeds the limitations of our machine's RAM, but we must ensure our batch size is large enough to minimise interpolation errors. We use the built in interp1 MATLAB function which performs a 1D linear interpolation for estimating values between data points. This finds values of a function underlying the data at intermediate points and requires the input x value, Time in our case, to be monotonic and unique.

Using a batch size of 10 and interpolation on 50000 points, we were able to interpolate the values to keep the input data within the machine's RAM size and minimise the error to an amount which was negligible. We preallocated matrix sizes for speed and rewrote the algorithm to reflect these changes.

- Preallocate space for grand sub-list.
- Read in batch of 10 files.
- Separate times and forces into two small matrix variables.
- Calculate list of force differences for each time ΔF.
- Append time and force difference ΔF to grand list.
- Determine average of forces over time.
- Interpolate on time and force differences to reduce 50000 points to 5000.
- Finalise average calculation of new data.
- Plot interpolated time with average force.

One issue with this method was that the x values (Time) were not unique. We solved this by adding a minor unique value to every time reading.

```
f = (1:length(Time))*1e-10;
```

This ensured the uniqueness of the values within the time matrix, thus allowing reliable interpolation to take place.

6.3 Experimental Results

With the plot of the initial million results with a medium compliant environment, we were able to see the expected graph depicting a logarithmically increasing force over time up to a maximum of 0.1s for which the KMC simulation was run.

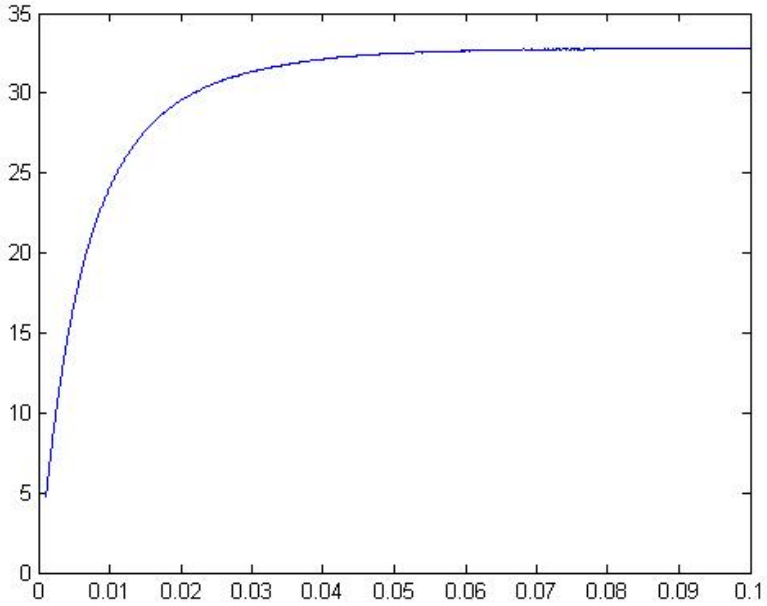

Figure 6.1: MATLAB analysis of 1,000,000 runs with medium compliance.

It is apparent in Figure 6.1 that some values are missing below 5pN on the x-axis. The reason for this is that the interpolation returned NaN (Not A Number) for times before the first event in the KMC simulation occurred. We rectified this small issue to establish a complete graph by replacing the NaNs with zeroes in the interpolation code.

```
IForce=interp1(Time+f', Force, ITime, 'linear', 0)/N;
```

This addition to the code ensured any values lying outside the range were set to 0, as well as keeping the time values in range unique so interpolation can be properly executed. Since the graph's curve is extremely smooth at 1,000,000 runs, we scaled down to analyse 250,000 with the applied fix.

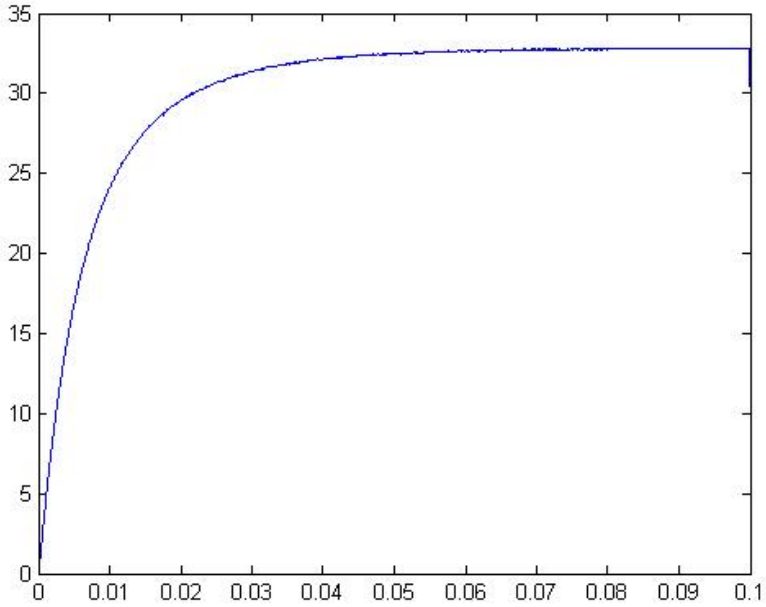

Figure 6.2: MATLAB analysis of 250,000 runs with medium compliance.

Figure 6.2 depicts the resulting plot from 250,000 runs of the simulation in an environment of medium compliance, using respective values for the actin and myosin: $Ca : 0.0002$ and $Cm : 0.0021$. Since the graph is still relatively smooth, this suggests that we can perform further research and analysis comfortably using a smaller number of simulation runs.

Figure 6.3 shows the resultant graph with the simulation of the crossbridges running under a less compliant environment, with the values fixed at 10x smaller than the medium compliance runs: $Ca : 0.00002$ and $Cm : 0.00021$. The maximum overall forces acting on the system were visibly higher than that within the graph in Figure 6.2 containing values relating to medium compliance.

In Figure 6.4, we can see the opposite effect. This shows the resultant plot from 250,000 runs of the crossbridge simulation in an environment concerning a high compliance, with values double those considered medium in Figure 6.2: $Ca : 0.0004$ and $Cm : 0.0042$. The effect of this is clearly visible and can be emphasised by overlaying the three graphs (small, medium and large compliances).

Figure 6.5 shows an overlay of the three graphs. From this we can see a clear difference between the forces that appear to depend on the environment's compliance. We can infer from this data that the lifetime of crossbridges is reduced in a more compliant environment, a claim that has never before been

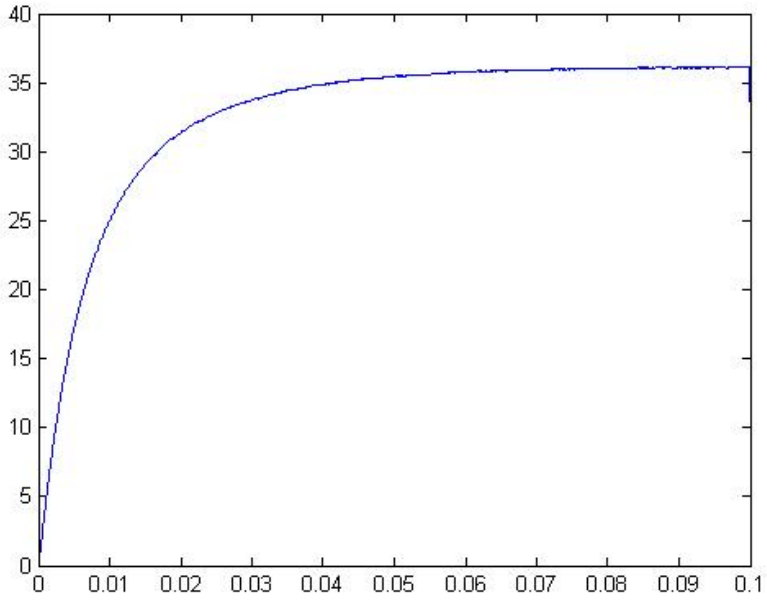

Figure 6.3: MATLAB analysis of 250,000 runs with low compliance.

tested. Since the data appears smooth when plotted for values as low as 100,000 runs, we can look ahead to conduct future research comfortably on a more modest number of runs.

6.4 Summary

In this chapter we have demonstrated that the results of our 1,000,000 simulation runs produced a graph which was more than sufficient to perform analysis and observe results pertaining to our hypothesis of whether compliant environments affect crossbridge (XB) activity. We therefore scaled down the simulation to an additionally sufficient 250,000 runs for three batches, each corresponding to different levels of actin and myosin compliance. These batches represented low, medium and high levels of compliance during the process of muscle contraction as described by the sliding filament model, and we produced plots of the Time and Force for each situation.

These plots required interpolation of the results, due to the quantity of the simulation outputs from the Condor distributed computing network. We present clear evidence that due to the graphs not being identical, an effect is bestowed upon the XBs as the environments changed. We can infer from the three differing graphs that since the maximum forces are lower in the high

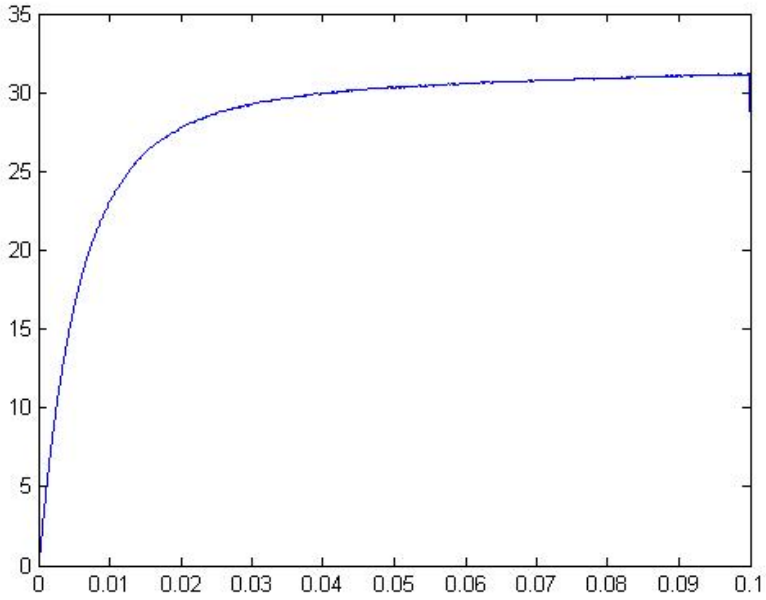

Figure 6.4: MATLAB analysis of 250,000 runs with high compliance.

compliant environments, that it is likely that the lifetime of an XB is lower because of this. This result has never been measured, and we believe this raises questions regarding why XBs behave in this manner and are affected by the compliance of their environments.

As a result, we present results which we intend to develop further and publish in order to understand more regarding the muscle contraction process. Due to our tests showing that 250,000 runs were sufficient for such analysis, we can comfortably rerun the simulation to provide extra data.

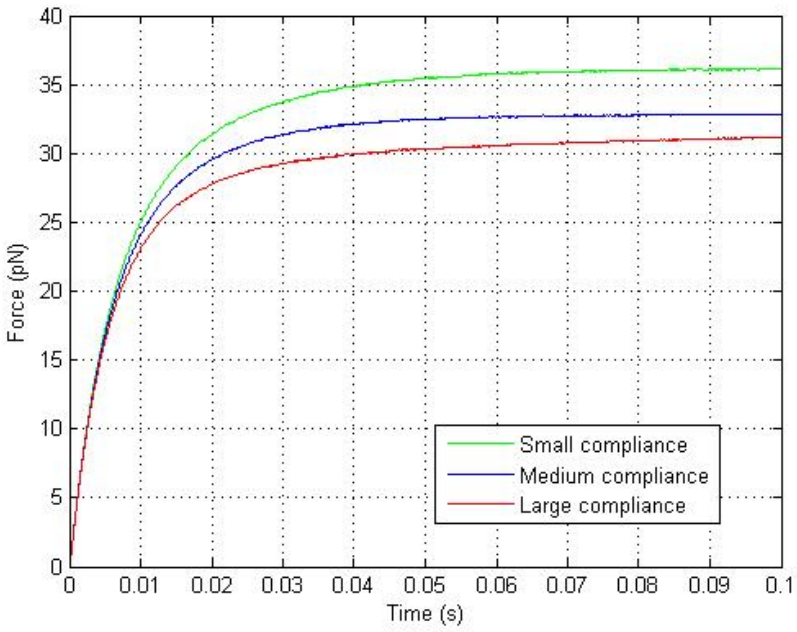

Figure 6.5: Overlay of MATLAB analyses depicting effects of compliance on force.

Chapter 7

Conclusions

Upon inception of this project, we set out to optimise and accelerate a MATLAB simulation of muscle contraction to allow a long term observation of 1,000,000 runs. The initial simulation could not be run in this manner feasibly, as we demonstrated using profiling and extrapolation that a sequential run would correspond to over four months for standard results and over a year if examining all time steps.

We have successfully produced an optimised version of the KMC simulation that was accelerated to run not only under our target projections of one day of computation time, but within 3 hours whilst saving all time points. This allowed us to run the simulation in batches of large amounts (250,000 runs) in order to observe the effects of different filament compliances on crossbridge (XB) activity. These calculations were computationally infeasible prior to this project, and we have made a forward step in the realm of understanding the process of muscle contraction further by way of Huxley's model.

7.1 Project Review

We present the reader with a reflection on our main achievements in this project. We have made the following contributions from the aims in chapter 1.

- An optimised version of the simulation in MATLAB in chapter 3. This allows the underlying KMC simulation code to be modified within a readable and user friendly environment, allowing for simpler maintenance of the fundamental code required to run the program. The code is optimised and ready for automated translation via the Simulink Coder, should this be necessary in future.

- An optimised version of the simulation in C++ in chapter 4. This approach was the key step in reducing the running time of the long term simulation to an amount that can allow the experimental results to be obtained in a reasonable time.

- An accelerated version of the simulation using High Throughput Condor in chapter 5. The measurements we present regarding Force/Time pairs at this point have never been calculated and are novel results. Consequently, any further analysis of these outputs are also novel, due to their availability being restricted by the underlying algorithm and methods before this report.
- Analysis of the previously unmeasured results which provide strong evidence that there is a relationship between the compliance of the actin and myosin filaments and the life of a crossbridge in chapter 6. Due to the unavailability of the simulation outputs since this method has never been tested, our MATLAB analysis of the results produce novel evidence upon which this claim can be based. Further research is required to confirm this and to find out why the relationship between compliance and force appears to exist, however. This has never previously been tested and we believe this report provides the first look into such measurements by drawing on means of computing optimisation and parallelism of the original algorithm.

We chose not to accelerate on the FPGA hardware or by way of GPGPUs due to various factors. This included the implementation complexity which would be introduced in the model, preventing it from being run on classic hardware as well as from being changed easily. This reduction of versatility would increase the difficulty in maintaining the code, and would significantly increase the development time. Furthermore, we have performed this project taking into account economic considerations. FPGA hardware is expensive and difficult to obtain. This report has presented a solution which parallelises the simulation using the open source and freely available Condor system that networks available computing resources of many types.

The optimised C++ version of the code is more versatile than the previous MATLAB code in the way that our wrapper uses a variety of functions to allow a range of simulations without changing hard-coded values. Use of the command line to provide inputs allowed the simulation to remain correct both in a standalone environment and in an accelerated situation where random number generation seeds became an issue due to parallel execution. We were able to modify the filament compliances between runs and examine the effects with fine control over the execution grain size.

These improvements to the algorithm produce a robustness which will allow for easier implementation of further research. Now that we have measured evidence of the XBs acting as a dependent system (as opposed to independent, as most studies assume), we can research the system further to understand why the XB lifespan is affected by the filament compliances. These novel results are publishable in their current form, and we aim to investigate the situation further by extending the simulation to include the state transitions describing the XB activity during the KMC method. By examining these transitions we can understand more about the process and aim to learn more about the sliding filament model.

7.2 Future Work

In this section we present possible avenues of further research that this project has enabled. Our optimisations have allowed us to run the simulation many times in a very short timeframe compared to the initial simulation, allowing future work to be carried out in reasonable times. Unfortunately, during the scope of this project, we did not have time to implement these further tasks. Potential extensions we leave as future exercises regarding this project include simulating more XBs to observe their activity and examine a more complex scenario with greater amounts of transitions. By increasing the amount of sites simulated, we can reduce the time during which we run the simulation. Since we have already shown results sufficient for analysis can be obtained with runs lasting no more than 30 minutes, we believe we can comfortably optimise and include more XBs in a modest timeframe. Running the simulation for 10x more XBs would result in only requiring the KMC method to run for 1/10 of its original time of 0.1s. We believe saving the state transitions as well as the forces under this new model will allow us to obtain more significant and novel results regarding the underlying biological system.

We may consider expanding this project for both medical and computing research.

7.2.1 Medical Research Aspects

The scientific significance of this project is high in that it implies that XBs treated as independent in simulations are overlooking an important factor in their ability to carry force. Since the compliance is shown to potentially affect their lifespan, we consider the following extensions and advancements for medical research from this project:

- Modelling larger groups of crossbridges simultaneously. This will allow potentially the revealing of more telling aspects of crossbridge behaviour during the simulation, and will require a lower run time in general. From this we can achieve a greater biophysical understanding of the interactions and effect of the compliances on the crossbridges.
- Saving crossbridge states as well as forces. This allows us an insight into the reasons why crossbridges behave in such a manner in environments with different compliances and will pave the way for more research into muscle contraction. At the greatest extent, we may learn information that may be key in assisting or treating muscle problems.

7.2.2 Computing Research Aspects

This project involves various problems faced in computing which occur during large simulation runs. We can focus on the following aspects to provide a real world view of how complex scientific processes can be optimised when implemented as mathematical models:

- Implementing the KMC simulation fully or partially on FPGA or GPGPU hardware. This will allow us to look into methods of speeding up the matrix inversion which takes place in order to produce significantly faster implementations than those described within this project, assuming I/O bottleneck issues are solved.
- Accelerating the average analysis on hardware to speed up graph generation. This will allow further speedup of the graph plots and interpolation, freeing up more resources to analyse more complex data in the same (or shorter) time.

7.3 Closing Remarks

Upon concluding this project we reflect on the aims we initially set as targets. We have successfully managed to accelerate, optimise and add flexibility and versatility to a stochastic model of muscle contraction at the fundamental level of muscle cells and filaments. We selected an arbitrary target of a million runs and expected to take a day's worth of processing time. Significantly outperforming these targets to run a million in under three hours allowed us to perform analysis that was not possible before. Analysis of the results obtained have generated novel measurements which we can use as a direction for further research. Our understanding of muscle contraction and crossbridge activity has been extended by our work, and we have gained additional knowledge from the experience of using high performance methods to speed up the algorithms involved.

We note that it would not have been possible to achieve the speedups we report with outdated hardware, and appreciate the performance gains granted from parallelism. These gains have inspired us to see the benefits and sheer potential of high performance hardware in accelerating scientific simulations to be able to model long term systems. Being able to harness such compute power in this form allowed us an insight into the future of research, where real-world models are represented more accurately by way of simulation rather than by using numerical solutions to solve complex equations. We understand the limitations of various types of supercomputing hardware, and while we accept that parallelism is not the solution to all problems, we postulate that without looking towards computing methods as a first resort for assistance in obtaining results to this biological problem, we would not have been able to break ground by obtaining such significant and novel results.

List of Figures

1.1 Amount of days required to complete various runs of the original simulation. 8

2.1 The arrangement of myofibrils in a muscle cell (Vander's Human Physiology). 12
2.2 The sarcomeric region is composed of proteins and is integral to muscle contraction and relaxation. 13
2.3 Actin and Myosin (thin and thick) filaments along with the myosin heads, troponin (TN) and tropomyosin (TM). 14
2.4 Relaxed and contracted states of a sarcomere structure. 15
2.5 A breakdown of complexity classes for both outcomes of P vs NP. 17
2.6 Number of transistors in various processors by year. The exponential growth corresponds to Moore's Law. 19
2.7 A control flow system (Courtesy of Maxeler). 21
2.8 A dataflow system (Courtesy of Maxeler). 22
2.9 3D Finite Difference Modelling relative speedup by DFEs (Maxeler). 23
2.10 A dataflow chip (Courtesy of Maxeler). 25
2.11 Dataflow graph for an increment function (Courtesy of Maxeler). 26
2.12 Dataflow graph for simple addition function (Courtesy of Maxeler). 27
2.13 Arithmetic calculations over time between GPUs and CPUs . . . 29

3.1 Actin (red) and myosin (blue) filaments, binding sites (circles) and connected XBs (vertical connecting lines) within the KMC simulation. 33
3.2 The reaction scheme showing kinetic states of the simulated system. 33
3.3 Amount of days required to complete various runs of the new simulation. 39

4.1 An LU decomposition of a 3x3 matrix. 46

5.1 Amount of hours required to complete various runs of the optimised C++ simulation. 52
5.2 Condor job queue state over a period of one year. 56
5.3 Condor state for 100,000 jobs running the KMC simulation. . . . 57

5.4 Condor job queue state over a period of one day. 57
5.5 Condor state for 25,000 jobs running the KMC simulation. 58

6.1 MATLAB analysis of 1,000,000 runs with medium compliance. . 63
6.2 MATLAB analysis of 250,000 runs with medium compliance. . . 64
6.3 MATLAB analysis of 250,000 runs with low compliance. 65
6.4 MATLAB analysis of 250,000 runs with high compliance. 66
6.5 Overlay of MATLAB analyses depicting effects of compliance on force. 67

List of Tables

2.1 Acceleration potential of dataflow engines over control flow systems. 21
2.2 A partial lineup of dataflow hardware offered by Maxeler. 25

3.1 An 8x3 sample of the 8x20001 matrix used in the KMC simulation. 34
3.2 MATLAB profile of the simulation as a script. 35
3.3 MATLAB profile of the simulation as a partially optimised function. 36
3.4 MATLAB profile of the simulation with LUT generation. 38
3.5 MATLAB profile of the simulation with 5000 time points. 38
3.6 C++ profile of the simulation with repeated LUT generation. . . 41

4.1 C++ profile of the simulation with once-only LUT generation. . 46
4.2 C++ profile of the simulation after final optimisations. 49

5.1 Granularity options for running the simulation on Condor. 54
5.2 Random number generator seed uniqueness between runs. 55

Bibliography

[1] A. F. Huxley, R. Niedergerke, *Structural Changes in Muscle During Contraction: Interference Microscopy of Living Muscle Fibres*, Nature 173, 971 - 973, 1954
doi:10.1038/173971a0.

[2] G. E. Moore, *Cramming more components onto integrated circuits*, Electronics Magazine (1965), ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf.

[3] NVIDIA, *NVIDIA Tesla Makes Personal SuperComputing A Reality*, http://www.nvidia.com/object/io_1227008280995.html.

[4] T. McCracken, *New Atlas of Human Anatomy China*, Metro Books 1-120, 1999.

[5] A. F. Huxley, A. L. Hodgkin, *A quantitative description of membrane current and its application to conduction and excitation in nerve*, Physiol, Aug; 117(4):500-44, 1952,
http://www.ebi.ac.uk/biomodels-main/BIOMD0000000020.

[6] H. Huxley, J. Hanson, *Changes in the Cross-Striations of Muscle during Contraction and Stretch and their Structural Interpretation*, Nature 173, 973 - 976, 1954
doi:10.1038/173973a0.

[7] R. Horowits, R. J. Podolsky, *The Positional Stability of Thick Filaments in Activated Skeletal Muscle Depends on Sarcomere Length: Evidence for the Role of Titin Filaments*, National Institute of Arthritis and Musculoskeletal and Skin Diseases, 1997
http://jcb.rupress.org/content/105/5/2217.full.pdf+html.

[8] L. Chin, P. Yue, J. J. Feng, C. Y. Seow, *Mathematical Simulation of Muscle Cross-Bridge Cycle and Force-Velocity Relationship*, Biophys, 3653-3663, 2006
doi:10.1529/biophysj.106.092510.

[9] A. F. Huxley, *Muscle structure and theories of contraction*, Biophys Chem, 7:255-318, 1957.

[10] R. C. Woledge, N. A. Curtin, *Energetic aspects of muscle contraction*, Physiol Soc, 41, 1985.

[11] S. A. Cook, *The complexity of theorem-proving procedures*, Proceedings of the Third Annual ACM Symposium on Theory of Computing, 151-158, 1971.

[12] W. I. Gasarch, *The $P \neq NP$ Poll*, SIGACT News 33(2): 34-47, 2002 doi:10.1145/1052796.1052804.

[13] V. Deolalikar, $P \neq NP$, HP Labs http://www.hpl.hp.com/news/2010/jul-sep/deolalikar.html.

[14] A. F. Huxley, L. E. Ford, C. Y. Seow, M. P. Slawnych, *A program for developing a comprehensive mathematical description of the crossbridge cycle of muscle*, Biophys, 1669-77, 1994.

[15] G. E. Moore, *Cramming more components onto integrated circuits*, Electronics Magazine, p4, 1965 ftp://download.intel.com/museum/Moores_Law/Articles-Press_Releases/Gordon_Moore_1965_Article.pdf.

[16] G. Estrin, *Reconfigurable computer origins: the UCLA fixed-plus-variable (F+V) structure computer*, IEEE Ann. Hist. Comput. 24, 3-9, 2002. http://dx.doi.org/10.1109/MAHC.2002.1114865.

[17] MathWorks, *Maximising Code Performance by Optimising Memory Access* http://www.mathworks.co.uk/company/newsletters/news_notes/june07/patterns.html.

[18] G. Xue et al, *Implementation of a Grid Computation Toolkit for Design Optimisation with Matlab and Condor*, Euro-Par 2003 Parallel Processing, Vol 2790/2003, 357-365 doi:10.1007/978-3-540-45209-6_54.

[19] M. Matsumoto, T. Nishimura, *Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator*, ACM Transactions on Modelling and Computer Simulation, 8(1): 3 - 30, 1998 doi:10.1145/272991.272995.

[20] K. Zeng, S. A. Huss, *Architecture refinements by code refactoring of behavioural VHDL-AMS models*, ISCAS, 2006.

[21] N. Kapre, A. DeHon, *Parallelizing Sparse Matrix Solve for SPICE Circuit Simulation using FPGAs*, IEEE International Conference on Field-Programmable Technology, December 2009 http://ic.ese.upenn.edu/pdf/matsolve_fpt2009.pdf.

[22] N. Galoppo et al, *LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware*, Proceedings of the ACM/IEEE SC/05 Conference, November 2005 http://gamma.cs.unc.edu/LU-GPU/.

[23] T. Nemeth et al, *An implementation of the acoustic wave equation on FPGAs*, SEG Las Vegas Annual Meeting, 2008
http://www.doc.ic.ac.uk/~wl/icprojects/papers/seg08max.pdf.