Implementing alternative polynomial multiplication algorithms and the AKS primality test in parallel

Vincent J. Renders
vr317@bath.ac.uk

Department of Computer Science
The University of Bath
September 2021

This dissertation is submitted for the degree of
Master of Science

Supervisor
Prof. James H. Davenport

On the implementation of the AKS primality test in CUDA

Submitted by Vincent J. Renders
for the degree of MSc in Computer Science
at the University of Bath

Copyright

Declaration

Abstract

This project aims to deliver two implementations of the Agrawal, Kayal and Saxena (2004) polynomial-time primality test in C++, both derived from a variant of AKS by Lenstra (2002). The first of these is the $Z_n[x]$ version, which uses the Number Theory Library (NTL) to deliver efficient polynomial multiplication using the classical algorithm, Karatsuba, Number Theoretic Transforms (NTT), and Chinese Remainder Theorem (CRT) or Fast Fourier Transforms (FFT). The second, denoted the Z version, uses the unconventional binary segmentation algorithm, on the recommendation of Crandall and Papadopoulos (2003).

Following a summary of the prerequisite mathematics, a review of the relevant literature, and a survey of the contemporary technology landscape, this dissertation implements the binary segmentation algorithm for multiplying univariate polynomials with integer coefficients in C++, alongside performance profiling to identify both benefits, if any, and areas of improvement therein. The runtime performance of Z is compared that of $Z_n[x]$, which uses more prominent asymptotically fast polynomial multiplication algorithms, and assessed in a wider context when applied to congruence testing; a key step of the AKS algorithm and significant contributor to its runtime.

As a final objective, an investigation is conducted into reducing the runtime of these implementations by exploiting their data-level parallelism to execute them, in parallel, on contemporary *NVIDIA* GPU hardware using the *CUDA* API. Ultimately, one seeks to accelerate the groundbreaking AKS algorithm through general purpose computing on graphical processing units (GPGPU) to the point which it becomes useful in practice and may be considered for real-world applications. To this end, the results of this investigation are presented for the attention of any person(s) undertaking this challenge in future and comprises basic parallel implementations on standard multi-core processors, parallel strategies for GPGPU, suitable number theoretic software libraries, and commentary on other general obstacles needing to be overcome before this may be achieved.

Contents

List of Figures

List of Tables

List of Algorithms

Acknowledgements

I would like to extend special recognition and gratitude to my project supervisor, Prof. James Davenport, for providing a unique and invaluable learning experience during challenging and unique circumstances in 2021. He has been patient with me as a naive student of his subject, opening my eyes to the beauty of primes and mathematics as well as enabling me to enjoy my final year in academia.

I would like to thank Dr. Thomas Powell, my personal tutor at Bath, for his support throughout the year, Dr. Tom Haines & Dr. Russell Bradford for putting up with the many questions I had about using *CUDA* or the *Hex* facilities. To the many friends and family members, who have taken it in turns to proof read and pick apart many a draft of this work: there are too many of you to express my thanks here at the length you each deserve, I am indebted to each of you.

I am ever thankful for my lovely girlfriend, Olivia, for all her support and encouragement from afar, whilst enduring the difficulties of your PhD at the same time. I can take you away on our well-earned, albeit belated, summer holiday at last. To my brother, Chris, thank you for allowing me to use your computer (your GPU more specifically) to test my programs, I could not have done it with out your help.

Finally, this project would not have been possible without the unwavering support from my parents for supporting me in Bath this year, before providing for, and putting up with, me while I have worked on this at home. There are not enough word for adequately expressing gratitude to you both for everything you have done for me.

This work is dedicated to my maternal grandfather and paternal grandmother, to whom I owe much of my work ethic and identity as a person from their significant contributions throughout my upbringing, and whom I very much wish could still be with us and be attend my graduation(s).

The problem of distinguishing prime numbers from composite numbers, and of resolving the latter into their prime factors, is known to be one of the most important and useful in arithmetic.

It has engaged the industry and wisdom of ancient and modern geometers to such an extent that it would be superfluous to discuss the problem at length.

Further, the dignity of science itself seems to require that every possible means be explored for the solution of a problem so elegant and so celebrated...

- Carl Friedrich Gauss

Disquisitiones Arithmeticae, 1801

Chapter 1

Introduction

1.1 Context

The primality testing algorithm published in the paper *PRIMES is in P*, by Agrawal, Kayal and Saxena (2004), also known as *the AKS test*, was the first, and remains the only, unconditional deterministic polynomial-time algorithm for primality testing. the discovery of an unconditional deterministic polynomial-time algorithm for primality testing proved the answer to a fundamental question in mathematical number theory and an enduring open problem in computer science: the decision problem of determining whether or not an integer, n, is prime (*PRIMES*) is in the complexity class ***P***, wherein ***P*** is the class of decision problems for which there is a deciding algorithm with polynomial time complexity (Bornemann, 2003). However, this groundbreaking algorithm, has not, as yet, found any practical significance in the realm of primality certification despite reductions in the complexity exponent following improvements and refinements from Lenstra and Pomerance (2019), and Bernstein (2003).

Much of the original work done to improve the AKS algorithm emerged in the early 2000s after the pre-publication the paper in which it was featured. Since that time, this work has stalled, receiving only minor algorithmic updates and revisions in recent years. Modifications to the original algorithm from Bernstein (2003, 2007) and Cheng (2007) at the time, followed by continual revisions of such work later on (Lenstra and Pomerance, 2019), have served to reduce the polynomial time exponent through algorithmic optimisations grounded in number theory. In practice, however, the complexity remains inferior to other primality testing algorithms (Brent, 2010). Early attempts from Bronder (2006), as well as Crandall and Papadopoulos (2003), at practically implementing the algorithm have been unsuccessful at delivering the computational runtime performance required for any application to be found.

Around the same time as these later works, came the advent of general-purpose (GP) computing on graphical processing units (GPUs) (Du et al., 2012). Since then, significant advancements in hardware for this type of computing have made parallel processing using modern general-purpose computing on graphics processing units (GPGPUs) more significant tool in scientific computing. Today, a single card can deliver up to 35 TFLOPS, according to vendors, compared to 228 GFLOPS in 2007, a 150-fold increase in single-precision (FP32) floating point performance in under 15 years.

The computational power and potential of GPGPU and application-specific integrated circuits (ASICs) has not been leveraged for primality testing in the way that it has been in artificial intelligence, for training machine learning models, or in distributed ledger technologies and blockchain applications, for the hashing algorithms used in transaction validation. The computational power afforded by GPGPU has propelled the rise of these technologies to the fore of technology, science, and, consequently, the mainstream media in the last five years. Exploration of such methods for primality testing is unprecedented and, therefore, warranted. Breakthroughs in this application may offer value to the popular public-key cryptosystem, RSA, which relies on very large primes, p and q, to generate secure keys (Laird, 2020). The security of the public key relies on the assumption that both p and q are definitely prime, thereby creating a vulnerability such that, should either one be a pseudo-prime and, crucially, *not* prime, any ciphertext encrypted using those values would be flawed.

In theory, the AKS test could be a useful verification step in this scenario and ensure the security of RSA by reliably certifying that the large prime factors, p and q, are definitely prime. In doing so, the intractability of factoring the large integer, and hence the integrity of the cryptosystem, is protected. Finding a suitable application would be a contribution of significance, however, this research would, at the very least, also serve to update experimental evidence on current hardware by offering benchmarks and comparisons. To this end, this project seeks to explore whether the AKS primality test can be made more practical and viable on modern high performance computing, or, more specifically, though general-purpose computing on graphical processing units.

The remainder of Chapter 1 formalises the proposed approach to the project, including hypotheses and methods. Some preliminaries in the literature review (Chapter 2) and technology survey in (Chapter 3) form the necessary theoretical foundations required for later continuation of the project. The former covers the fundamental mathematics related to algorithms and primality testing, as well as providing an analysis of the literature surrounding the AKS test and its place in the context of primality testing, followed by an evaluation of contemporary hardware and software technologies related to this project in the latter. Chapter 4 refines the research objectives and proposes the deliverables for this research. To this end, Chapter 5 and Chapter 6 give an account of the research conducted, with results and interpretations. Finally, this research is critically evaluated and concluded in Chapter 7.

1.2 Theoretical Framework and Hypotheses

Despite being a significant theoretical breakthrough, the AKS primality test currently has no practical applications, meaning it is considered a *galactic algorithm*. It stands to reason then, that a sensible initial hypothesis to test, or challenge in this case, might be:

> 'The AKS primality test is impractical and, as such, has few, or no, real world applications, even when implemented on modern computer hardware.'

Furthermore, the AKS algorithm is limited in the way that it cannot produce a primality *certificate* that can verify the result. Consequently, a further hypothesis to test could be:

> 'A parallel implementation of the AKS primality test on modern hardware cannot facilitate the output of a certificate of primality.'

Definition 1.2.1. In complexity theory and computing, a certificate is a concept which is analogous to a *witness* in mathematics, whereby a set of states or values is produced during a computation which allow the solution path to be verified. A useful certificate should be compact and concise such that its verification is a substantially cheaper computation than that which originally found the solution. In the context of primality testing, algorithms might produce a witness to primality or compositeness depending on whether the input is prime or not prime respectively.

Attempting to challenge one or both of these hypotheses is an ambitious task. Nevertheless, a method for doing so is described below to shed further light on the problem and whether a practical application for AKS primality test can be found.

1.3 Proposed Methods

The following general approach is proposed:

1. Write a basic initial implementation of a chosen variant of the AKS primality test in a standard programming language for validation and benchmarking purposes.
2. Refine this initial algorithm using other primality algorithms in combination with AKS and utilising suitable supporting libraries to incorporate faster multiplication algorithms, such as Karatsuba's method (Davenport, 2021), the Schönhage and Strassen (1971) algorithm based on Fast Fourier Transforms (FFT), and others.
3. Implement this refined algorithm in the vendor-based *CUDA* API for implementation on a modern *NVIDIA* multi-core GPUs using parallel processing.
4. Assess the performance and practicality of the AKS algorithm against other primality testing algorithms.

As each step builds on the last, it is hoped that this method is suitable and will gradually produce a complete and robust evaluation of the hypotheses. Under COVID-19 restrictions, the method is feasible and enabled by remote communication, remote access to the hardware facilities required, and the personal facilities available.

As with any any opportunity for novel research, there relatively little in the way of literature and guidance specific to the contemporary hardware and software, available to the author at the time of writing, for this application. A revised and more informed approach may be found following further investigation in the literature review and technology survey in Chapter 2 and Chapter 3 respectively.

The findings obtained from this method can be validated for consistency and rationality using existing data and methods. Where there is no existing comparable data to validate results, additional steps could be taken to generate it. Otherwise, it could (or perhaps should) be left open to others to review or contribute to in future.

Chapter 2

Literature Review

This section outlines the fundamental mathematical concepts relevant to this project, including; computational complexity, algorithms, a background of the millennia-old exercise of primality searching and testing, the importance of AKS in this context, as well as an examination of the algorithm and its computational implementations over the years. Before doing so, the basics of number theory in the context of AKS, and primality testing generally, must be covered, starting with a definition of prime numbers and their importance to a technology- and internet-driven civilisation. Matters relating, but not limited, to advanced formal proofs of correctness of the algorithms and concepts referred to here, their complexity, or lemmas upon which they are based are deemed beyond the scope of this work.

2.1 Mathematical Foundations

2.1.1 Number Theory & Primes

Number theory is the branch of pure mathematics dedicated to the study of integers. *Natural numbers*, $\mathbb{N}$, are the set of positive, or non-negative, integers, including zero. *Prime* numbers are a subset of the natural numbers, and are formally defined as follows.

Definition 2.1.1. A prime number, *p*, is a natural number, greater than one, whose only factors (divisors) are one and itself. Let the set of prime numbers be denoted $\mathbb{P}$ such that:

$$\mathbb{P} = \{p \mid p \in \mathbb{N}, p > 1, p \text{ is prime}\} \tag{2.1.1}$$

The number one, by convention, is not considered prime, but rather a *unit*. A natural number, a which is neither a unit nor a prime, which is to say it has two or more factors greater than one, is *composite* (Davenport, 2008). A basic, but laborious, test for determining whether or not a natural number, n, is prime would be to verify if any other natural number, d (where $1 < d < n$) is a factor of n. This is called *trial division* and the number of division operations required in this search is proportional to n. However, it is only necessary to search with odd integers up to, and including, the positive root of n, $\sqrt{n}$, since the the largest factor of n must be $n/2$ and each possible combination of factors of n up to $n/2$ appears twice. An extremely trivial demonstration of this corollary is given as follows:

Let $n = 36$.

It follows that $\frac{n}{2} = 18$.

The pairs of factors of n are 2×18, 3×12, 4×9, 6×6, 9×4, 12×3, and 18×2.

Therefore, one need only test primes for even divisibility up to, and including, $\sqrt{n}$ $(= 6)$, since factors begin to repeat after this point.

The notion that a composite number can be expressed as a product of factors gives us the *fundamental theorem of arithmetic* which was first stated and proved by Gauss in 1801.

Theorem 2.1.1. *Any natural number can be expressed as the product of its prime factors in one, and only one, way (Davenport, 2008).*

$$n = \prod_i p_i^{\alpha_i} \tag{2.1.2}$$

One of the significant implications of this theorem is to do with the process of factoring and its contextual relevance to primality testing. Factorisation and division, as well as other arithmetic operations, of natural numbers is relatively trivial when the numbers are small, however, for large numbers, it becomes another matter entirely. In this instance, due to the contextual relevance to cryptography, a *large number* is generally defined as any positive integer with a length in the thousands of binary digits (or hundreds of decimal digits), so greater than 2^{1024} $(\approx 10^{308})$. The process of factorising integers of this size is inherently difficult and computationally infeasible. This is known as the *factorisation problem* and it underpins the integrity of RSA-based public-key cryptography, which uses the product of two different primes of this size, p and q, to generate an even larger *semiprime* number as part of the public key (Laird, 2020).

Definition 2.1.2. A semiprime number, *n*, is a natural number and the product of exactly two primes, which may be equal.

A *semiprime* is not the only existing subclass of prime number, there are a multitude of other classes which categorise the primes in seemingly every conceivable way. There are, among the more notable of these, the *Mersenne primes*, some of the largest known primes with lengths of 10^8 decimal digits, and *Sophie Germain primes*, or *safe primes*, which are highly relevant to the AKS primality test and are discussed at greater length in Section 2.4.

A further implication of the *fundamental theorem of arithmetic*, directly relevant to the AKS algorithm, is that the notion of a unique product of primes enables direct comparison of two or more integers in terms of their common divisors, giving rise to two fundamental properties; the *greatest common divisor*, which is also known as the *highest common factor* and abbreviated to the *GCD* or *HCF* accordingly, and the *least common multiple*, which is abbreviated to the *LCM*.

Definition 2.1.3. The greatest common divisor, or highest common factor, of two natural numbers, m and n, is largest number which occurs as a factor in both m and n, and can be expressed as a product of their common prime factors taken to the highest power to which it divides them *both*.

Definition 2.1.4. The least common multiple of two natural numbers, m and n, is the product of their common primes taken to the highest power to which they occur in *either* number.

Computing the greatest common divisor of two integers, n and r, is an operation required in a the steps of the AKS algorithm, as examined later in Section 2.4. There exists an algorithm for efficiently obtaining it, called the Euclidean division algorithm, or just Euclidean algorithm, which is still used by many of the modern software libraries for computational number theory due to its apparent simplicity. This algorithm is more formally described by the computationally-syntactic pseudocode below.

Algorithm 2.1.1 The Euclidean division algorithm

```
1: procedure $GCD(a, b)$    ▷ Finds the greatest common divisor of natural numbers $a$ and $b$
2:     $r := a \pmod{b}$
3:     while $r \neq 0$ do    ▷ Iterative loop ends when $r$ reduces to zero
4:         $a := b$
5:         $b := r$
6:         $r := a \pmod{b}$
7:     return $b$    ▷ The greatest common divisor is $b$ when $r$ is zero
```

Two further noteworthy comparative properties of two integers, obtainable from the fundamental theorem and subsequent unique product of primes. The first is *co-primality*, which occurs as a consequence of two integers having no common factors, and subsequently gives rise to the *Euler totient* or *Euler phi* function.

Definition 2.1.5. Two natural numbers, m and n, are co-prime, or relatively prime, if they have no common divisor except one.

Definition 2.1.6. The Euler totient, or phi, function,$\phi(n)$, returns a count of the positive integers, x, less than n which are co-prime, or relatively prime, to n.

$$\phi(n) = |\ \{x \in \mathbb{N} \mid 1 \leq x < n, GCD(x, n) = 1\}\ | \tag{2.1.3}$$

Euler's totient is applied in *Carmichael's lambda function*, or *reduced totient*, which is, in turn, especially relevant to both the RSA encryption algorithm and, as the reader will soon see, primality testing.

Definition 2.1.7. Carmichael's lambda, or reduced totient, function is the smallest positive divisor of Euler's totient which satisfies Euler's theorem. If n is a positive integer, $\lambda(n)$, is defined as the smallest positive integer m such that $a^m \equiv 1 \pmod{n}$ for all a such that $GCD(a, n) = 1$. To compute $\lambda(n)$, n must be expressed as a product of its prime factors (Equation 2.1.2) and calculated recursively using the following formulae:

$$\begin{aligned}\lambda(n) &= \lambda(p_1^{\alpha_1} \times \cdots \times p_i^{\alpha_i}) \\ &= LCM(\lambda(p_1^{\alpha_1}), \ldots, \lambda(p_i^{\alpha_i}))\end{aligned} \tag{2.1.4}$$

where the p_i are distinct positive prime numbers, the exponents $\alpha_i \geq 0$, and

$$\lambda(p^\alpha) = \begin{cases} \phi(n) & \text{if } \alpha \leq 2 \text{ or } p \geq 3 \\ \frac{1}{2}\phi(n) & \text{if } \alpha \geq 3 \text{ and } p = 2 \end{cases} \tag{2.1.5}$$

Moving away from comparing two integers by their relative primality, the final concepts to define in this part, included here due to their direct relevance to the AKS algorithm, are the *perfect power* and the *prime power*, a special case of the perfect power when $m \in \mathbb{P}$.

Definition 2.1.8. A perfect power, n, is an integer formed from a product of a single integer, m, i.e. it may be resolved into equal factors, or divisors. This means it has an exact root and n can be expressed as a perfect k^{th} root.

$$n = m^k$$
$$m = \sqrt[k]{n}$$

Definition 2.1.9. A prime power, n, is an integer formed from a product of a single prime, p, or a positive integer power of p, i.e. all its factors, or divisors, are equal and prime. As before, it has an exact root and n can be expressed as a perfect k^{th} root.

$$n = p^k$$
$$p = \sqrt[k]{n}$$

Much like the rest of the concepts in this section, this final definition is inherently simple. None of the definitions in this introductory section are far beyond the abilities of primary school children, however, it is useful, and essential, to become formally reacquainted with such fundamentals. Each of them is astutely applied to construct the momentous AKS algorithm, and it is this simplicity aspect, in particular, to which mathematicians attribute the beauty of the algorithm.

2.1.2 Complexity Theory

Time complexity is a standard measure for generalising the amount of time taken for a function or an algorithm to return an output based on the size of the input. Given that the time (number of steps) is proportional to the size of input, complexity is essentially a function which takes the size of an input, n, and returns the number of steps. By taking the upper bound, or worst case, complexity of the function or algorithm and focusing on the *order of growth* in the number of steps, this is formalised using 'Big-O' notation (Powell and Vorobjov, 2020). It is formally defined in the context of the positive integers, $\mathbb{Z}_+$.

Definition 2.1.10. For two functions $f, g : \mathbb{Z}_+ \to \mathbb{Z}_+$, we write that $g(n)$ is approximated by $f(n)$ for big numbers, if there exists constants $n_0 > 0$ and $c > 0$ such that $f(n) \leq cg(n)$ for all $n \geq n_0$:

$$f(n) = O(g(n)) \tag{2.1.6}$$

Big-O notation is useful for comparing and analysing the efficiency of different algorithms without having to formalise them in detail. In cases where there are many occurrences of the binary logarithm (logarithm base 2, $\log_2 n$, henceforth denoted $\log n$), such as for the conversion of an input into binary digits (Brent, 2010) when algorithms are run on a computer or indeed any other valid reason for such a case, it is customary to use a variant of this notation to approximate them. This is called 'soft-O' notation, denoted $\tilde{O}$ and defined below.

Definition 2.1.11. For two functions $f, g : \mathbb{Z}_+ \to \mathbb{Z}_+$ and some constant $k \in \mathbb{Z}_+$ we can rewrite (Equation 2.1.6) as (Equation 2.1.7) and express it in the form seen in (Equation 2.1.8).

$$f(n) = O(g(n) \log^k g(n)) \tag{2.1.7}$$
$$\equiv \tilde{O}(g(n)) \tag{2.1.8}$$

The highest order term in the expression for the number of steps, $g(n)$, determines which of the numerous complexity classes to which the function or algorithm belongs. A non-exhaustive selection of these classes is included in table Table 2.1 with a generalisation of their respective notations. It is generally accepted convention that an algorithm which runs to completion in, up to and including polynomial time complexity, i.e. $f(n) = O(n^d)$ for a constant $d > 2$,[1] is described as *computationally feasible* or *tractable*.

Definition 2.1.12. An algorithm which is *computationally feasible* or *tractable* is one which may be used for practical computations for reasonably sized inputs (Powell and Vorobjov, 2020).

Computational problems themselves also have complexity classes, whereby they are classified by the time complexity of the most efficient known algorithm which solves them. One of these, P, is defined below. There are many open problems which either have no algorithm which solves them, or no *efficient* algorithm rendering it feasible. Complexity theory is the field of theoretical computer science which studies such algorithms and searches for efficient solutions to open problems.

Definition 2.1.13. The class of decision problems for which there is a deciding algorithm with polynomial complexity is called P (Bornemann, 2003; Powell and Vorobjov, 2020).

Class Name	Notation
Constant	$O(1)$
Logarithmic	$O(\log n)$
Linear	$O(n)$
Quadratic	$O(n^2)$
Polynomial	$O(n^d)$
Exponential	$O(c^{p(n)})$
Factorial	$O(n!)$

Table 2.1: Time complexity classes and the associated number of steps in Big-O notation, where n is number of steps proportional to the size of the input, c and d are integer constants, and $p(n)$ is a polynomial. Intuitively, as $n \gg c$, factorial time may grow much faster than exponential time.

[1]Strictly speaking, exponents $d = 1$ and $d = 2$ are admissible, and are, therefore, also considered *polynomial* time. In this dissertation, these cases are treated as distinct sub-classes for the purposes of completeness and consequently classified as *linear*- and *quadratic*-time respectively (Table 2.1) according to the magnitude exponent. By this logic, a complete list of such cases would also include the other polynomial forms such as *cubic*, *quartic*, and higher. However, for an elementary treatment of the topic, this is implied by the inclusion of linear and quadratic sub-classes alone.

2.1.3 Modular Arithmetic

Modular arithmetic is a system of integer arithmetic whereby numbers cycle around a fixed quantity, known as the modulus. A modulo operation can be described as a floored division in which the quotient, q, is discarded and the least non-negative remainder, r, is the output. The most conspicuous and intuitive use of modular arithmetic in day-to-day life is the 12-hour clock, where one reads the time as the remainder of the hour shown with a modulus of 12 (e.g. 6:00 is the remainder of 18:00 modulo 12:00).

$$q = \lfloor a/b \rfloor \tag{2.1.9}$$
$$a = b \times q + r \tag{2.1.10}$$
$$r = a - b \times q \tag{2.1.11}$$

There are two common forms of notation for modular arithmetic. The first can be described as *inline*, and uses an *operator* in the same way that one might do to denote any other arithmetic operation (such as multiplication, $\times$, or addition, $+$). The second may be likened to a 'postfix' notation, whereby the moduli are included on the opposite side to the numbers or expressions they operate upon.

$$23 \bmod 21 = 2$$
$$23 \equiv 2 \pmod{21}$$

An important distinction between the two cases is the equivalence notation used; the mathematical equivalent ($=$) is strictly reserved for the first case whereas congruent (or in the same equivalence class, and denoted by $\equiv$) is the only correct option in the second case, as the two equivalent examples above demonstrate. The second form can also be used to show two integers, a and b, as congruent modulo n if they have the same remainder when divided by n:

$$a \equiv b \pmod{n}$$
$$98 \equiv 23 \pmod{3}$$

Modular arithmetic also extends to polynomials having real integer coefficients. The notion of dividing by the modulus to give the remainder is unchanged and the classical method for obtaining a remainder when dealing with polynomial moduli is long division. The example below demonstrates how integer moduli are applied to polynomials. In cases such as this, the remainder operations only apply to the integer coefficients, but may still have an affect the degree of the polynomial.

$$\text{Let } f(x) = 24x^6 + 8x^5 - 12x^4 - 4x^3 + 21x^2 + 10x + 1$$
$$f(x) \equiv x^2 + 1 \pmod{2}$$
$$f(x) \equiv x + 1 \pmod{3, x^2 + 1}$$

Integer and polynomial moduli may be applied in combination and modular operations involving integers *and* polynomials are featured in the AKS test. It is also worth noting that, since multiplicative operations are commutative, the order of multiple modulo operations is irrelevant; $x \pmod{a, b}$ is the same as $x \pmod{b, a}$, where x, a, and b may be integers, polynomials, or any combination of both. In practice, however, the order of the operations may have influence the runtime of programs implementing them computationally, and should therefore not be neglected entirely.

2.2 Computational Arithmetic

From the last section, it is clear that achieving polynomial time complexity, $O(n^d)$, for any algorithm is critical to its computational feasibility. This applies to operations as seemingly trivial as addition, subtraction, division and multiplication, since they must also have algorithms which compute them as subroutines. As such, optimising 'simple' operations such as these can have cascading effects on the overall time complexity of an algorithm, particularly when the numbers become very large, as discussed. This section concerns itself itself with a selection of methods which accelerate the 'traditional' or 'classical', $O(n^2)$ time, methods for multiplying large integers and polynomials. Both of these cases are important to different variants of the AKS primality test and, thus, pertinent to any computational implementation thereof.

2.2.1 Karatsuba's Algorithm

The works of Toom (1963) and Cook (1966) show that integer multiplication, division, and greatest common divisor operations can be done in 'essentially linear time' (Bernstein, 2004) although, strictly speaking, this is said to be more like quasi-linear time ($\tilde{O}(n)$). This algorithm works by splitting the integers being manipulated into a number of parts, each of a certain length.

In practice, the overhead costs of this algorithm mean it is slower than the traditional algorithm for small integers and, for large integers, the Schönhage–Strassen algorithm exceeds it in performance (Crandall and Pomerance, 2006). Only a select few software libraries include this algorithm, with most favouring the likes of Karatsuba and FFT-based algorithms, such as Schönhage–Strassen, instead.

The multiplication algorithm from Karatsuba and Ofman (1963) is related to Toom-Cook methods in the way that methods may be applied on integers, and best used for integers in an intermediate range. Equation 2.2.1 and Equation 2.2.2 show how the operations for computing a product of two linear polynomials compares with the traditional methods and that of Karatsuba respectively. The number of multiplications is reduced from four to three, however, the number of addition operations has gone up from one to four.

$$(aX+b)(cX+d) = acX^2 + (ad+bc)X + bd \tag{2.2.1}$$
$$(aX+b)(cX+d) = ac(X^2-X) + (a+b)(c+d)X + bd(1-X) \tag{2.2.2}$$

Although polynomials are used in this example from Davenport (2021), the principle is the same for integers, with varying reports for the time complexity of this method: $O(\ln^{\log 3/\log 2} N) \approx O(\ln^{1.58} N)$ for a size-N multiply (Crandall and Pomerance, 2006), or $O(n^{\log_2 3 \approx 1.585})$ for arbitrary numbers of n terms (Davenport, 2021).

This is *asymptotically faster* than the traditional method and is widely used in software libraries to reduce the complexity of subroutines in primality tests by optimising arithmetic operations on large integers and polynomials. However, there remain algorithms which are asymptotically faster still, based on fast Fourier Transforms.

2.2.2 Fast Fourier Transforms & the Schönhage–Strassen Algorithm

Formally, the fast Fourier Transform (FFT) is an algorithm for efficiently computing the Fourier Transform of digital wave forms. In the literature, the term fast Fourier Transforms, or FFTs, is, somewhat inaccurately, synonymous with the algorithm of Schönhage and Strassen (1971), an asymptotically fast multiplication algorithm which is based on recursive use of fast Fourier Transforms, in the proper mathematical sense. Also known, maybe more accurately, as Number Theoretic Transforms (NTT), Schönhage and Strassen (1971) generalise discrete Fourier transform to finite fields, showing it to be the fastest known method of multiplying two large integers at the time.

Granville (2004) describes FFTs as the most efficient algorithm for multiplying two integers a and b, achieving $O(u \log u \log \log u)$ time, where u is the length of longest of the two when expressed in binary, $|\max(a, b)|$. Davenport (2021) writes this as $O(\max(a, b) \log \min(a, b))$, and highlights the convention of only applying this particular method when the binary length, b, is large and instead using Karatsuba's method for the intermediate range $16 \leq b < 4096$ and any other, still polynomial-time, method for any small $b < 16$. A convention, such as the one described here, may be called a *polyalgorithm* according to the definition below.

Definition 2.2.1. A polyalgorithm is a set of algorithms used with accompanying conditions for defining which is used and when.

This idea of varying the method according to the domain of the integers a and b is applied in many of the computational arithmetic and number theoretic software libraries discussed in Section 3.3. Takahashi (2010) used Karatsuba and Schönhage-Strassen methods in combination to overcome the memory requirements of the latter, a technique which may be useful should the same problem be encountered when implementing AKS in parallel on GPUs.

Although the Karatsuba and Schönhage-Strassen algorithms are popular in software libraries, and were the fastest around at the time Granville (2004) and Dietzfelbinger (2004) published their work, the best asymptotics in the literature were published by Harvey and Van der Hoeven (2021). The work describes recently achieved integer multiplication in $O(n \log n)$ time using FFT-based methods. Computational implementations of this method, if they already exist, could be premature, therefore the algorithms, pre-included in existing libraries and known to be sound, are perfectly adequate and will be taken forward. This is unlikely to change and implementation of the method from Harvey and Van der Hoeven (2021) will be left to another, for future work, unless a robust implementation becomes available, in which case experimentation may prove worthwhile.

2.3 Primality Testing & AKS

The exercise concerning the search and testing of primes is one which has entertained mathematicians for centuries, if not millennia. *The Sieve of Eratosthenes* from the 3rd century BC is widely accredited as being the first algorithm related to primes, generating all those between one and a chosen integer, n, in $O(n \log n)$ time (Goldwasser and Kilian, 1999). Primality testing returned to popular mathematics once more in the 17th century AD, undertaken by the likes of Fermat and Gauss. Progress in the field has stagnated somewhat since, whilst primes have arguably become even more significant to the everyday lives of humans.

This section addresses the progress of modern mathematics and the landscape of primality testing algorithms before AKS by covering a selection them, chosen either for their significance to the field or for their relevance to AKS and this project. The relevant technical concepts vary between each algorithm and are, in many cases, far more advanced than the elementary topics covered in Section 2.1. They are, nevertheless, covered in sufficient depth where necessary. In computing, *randomised* algorithms are typically classified either as a *Las Vegas* or *Monte Carlo* algorithm.[2] Table 2.2 summarises a selection of primality testing algorithms before addressing each one in turn.

Definition 2.3.1. A randomised algorithm is one which incorporates a certain amount of randomness as part of its operation, i.e. a given input may pass through a random sequence of states before arriving at the solution, or through a deterministic set of states given a random seed value.

Definition 2.3.2. A Las Vegas algorithm is a randomised algorithm which always halts and has an error rate of zero, however, runtime may be unpredictable as it varies depending on the input.

Definition 2.3.3. A Monte Carlo algorithm a randomised algorithm whose output has a non-zero (typically small) probability of error.

Name	Class	Time	Conditions	Certificate
Fermat	Probabilistic	$\tilde{O}(k \log^2 n)$	None	Compositeness
Miller	Deterministic	$O(\log^4 n)$	ERH is *True*	Compositeness
Miller-Rabin	Random	$\tilde{O}(\log^3 n)$	None	Compositeness
Baillie-PSW	Deterministic	$O(\log^{2+o(1)} n)$	$\|n\| < 2^{64}$	Primality
Baillie-PSW	Probabilistic	$O(\log^{2+o(1)} n)$	$\|n\| > 2^{64}$	Primality (*conj.*)
APR	Deterministic	$O(\log^{c \log\log\log n} n)$	None	Primality
ECPP	Random	$O(\log^{4+o(1)} n)$	None	Primality
AKS	Deterministic	$\tilde{O}(\log^6 n)$	None	Primality

Table 2.2: Tabulated comparison of the selected primality testing algorithms, listed only in the order they appear in this section. *Conj.* is an abbreviation of *conjecture* and indicates conjectural certification.

[2]Etymology: these names refer to the grand casinos in the respective cities, but are also a nod to the element of 'gambling' involved with each kind; try random options, one by one, from a set of possible solutions until you get lucky - *Las Vegas* - or, attempt a procedure a single time and accept that there is a (small) possibility it could be wrong - *Monte Carlo*.

2.3.1 Fermat's Little Theorem

Theorem 2.3.1. *Fermat's Little Theorem proposes that, for all $1 \leq a \leq p$ where p is the integer being tested for primality, the following assertion holds:*

$$a^p - a \pmod{p} \equiv 0 \tag{2.3.1}$$

Pierre de Fermat, proposed a theorem, given above, about about primes. An algorithm which tests this theorem, such as that in Algorithm 2.3.1, would accurately determine whether or not an input p is prime until it reaches the number 561, the first *Carmichael Number*. The number 561 is, in fact, *not* prime; it is equal to the product of $3, 11$ and 17, yet it would still satisfy Fermat's theorem for all values of a (Aaronson, 2003).

Algorithm 2.3.1 An algorithm for Fermat's Little Theorem

```
1: procedure $FLT(p)$                          ▷ Returns True if an input $p$ is PRIME
2:     for $a = 1$ to $p$ do                     ▷ Loop for all $1 \leq a \leq p$
3:         if $a^p - a \pmod{p} \neq 0$ then
4:             return $False$          ▷ $p$ is NOT PRIME if condition met for any $a$
5:     return $True$                               ▷ Otherwise, $p$ is PRIME
```

Whilst Fermat's little theorem is flawed, it remains useful to the extent that it gives an indication of the likelihood that an integer is prime and is, therefore, known as a *probabilistic* primality test. The idea that a composite number can pass a primality test also gives rise to the concept of *pseudoprimes*.

Definition 2.3.4. A probabilistic primality testing algorithm is one which can only decide whether a given integer is *LIKELY PRIME* or definitely *NOT PRIME*.

Definition 2.3.5. A pseudoprime is a composite number falsely designated as prime by a primality test, i.e. it passes all the steps in the algorithm and the answer returned is a false positive.

The converse of Fermat's theorem, that $a^n - a \pmod{n} \neq 0$ for any integer n, is also useful as a *proof of compositeness* of n, without having to find its prime factors (Brent, 2010; Granville, 2004). Fermat's theorem is one of the simplest and best known tests which, despite only being a probabilistic test, has influenced the many other primality tests ensuing it.

2.3.2 The Miller-Rabin Primality Test

Miller (1976) proposed an algorithm which proves primality in *deterministic* $O(\log^4 n)$ time and assumes the *Extended Riemann Hypothesis*. This hypothesis is yet to be proven (Caldwell, 2021; Brent, 2010), however. An exact definition of the *Extended Riemann Hypothesis* (ERH) is beyond the scope of this work, it is enough to understand that, should the ERH be proven true, Miller's test would be one of the most, if not *the* most, efficient deterministic polynomial time algorithm for testing primality.

Definition 2.3.6. A deterministic algorithm is one which, for a given input, will pass through the same sequence of states each time before reaching the same solution.

Rabin (1980) removed this condition by giving a *randomised* version of the test from Miller (1976), compromising its determinism and converting it to a *Monte Carlo* in doing so. Using an extension of a result from Fermat's theorem, a necessary, but insufficient, condition for the primality of an integer, n, is extracted (Dietzfelbinger, 2004).

The resulting unconditionally randomised, probabilistic polynomial-time algorithm is extremely fast and feasible for numbers with 10^6 decimal digits Brent (2010), but with it comes a minute error probability of error. Just as with Miller's original algorithm, Rabin's adaptation becomes deterministic if the ERH is proven true, as does a similar Monte Carlo algorithm from Solovay and Strassen (1977) (Dietzfelbinger, 2004).

2.3.3 The Adleman-Pomerance-Rumely & APR-Cohen-Lenstra Tests

The APR primality test, named after its inventors, Adleman, Pomerance and Rumely (1983), differs from the other tests covered so far in the sense that it is a deterministic almost-polynomial (exponential) time algorithm, testing the primality of an integer n in $O(\log^{c \log\log\log n} n)$. Cohen and Lenstra (1984) developed this test further (APR-CL) and implementing it practically so that it could successfully test integers in the 100 decimal digit range for primality in seconds (Caldwell, 2021; Cohen and Lenstra, 1987) but, ultimately, failed to get it down to polynomial time. Interestingly, this test may be found in the PARI/GP software library as part of a polyalgorithmic implementation.

2.3.4 The Baillie-PSW Primality Test

The Baillie-PSW (BPSW) primality test is something of a hybrid of a few different algorithms, originally developed by Baillie and Wagstaff (1980) and later refined by Pomerance, Selfridge and Wagstaff (1980). It is a Monte Carlo algorithm which uses Jacobi symbols and is based on strong Euler probable primes, which is yet another extension of Fermat's little theorem (Caldwell, 2021). This particular test is said to conjecturally certify primality in $O(\log^{2+o(1)} n)$ time (Bernstein, 2004). Pomerance (1984), however, deems this conjecture to be implausible for very large n.

Irrespective of conjecture, it has been empirically shown that this test is deterministic and polynomial time for all inputs less than 2^{64} ($\approx 1.85 \times 10^{19}$). This is to say, there are no composite numbers below this number that would be able to evade the algorithm as pseudoprimes, which is perhaps the single most important implication.

The significant implication of this is that it may be used to validate any new implementations of primality testing algorithms; an implementation of AKS in parallel could be verified using this algorithm and inputs of a limited size to protect against programming errors. Furthermore, the BPSW test is used in many software libraries as a basic computational primality test for integers within a certain range.

2.3.5 Elliptic Curve Primality Proving

Most are familiar with elliptic curves in the context of elliptic curve cryptography (ECC), the second major public-key cryptographic method, alongside RSA cryptography, however, elliptic curves also have uses in factorisation (Lenstra, 1987), and, consequently, primality testing (Goldwasser and Kilian, 1986, 1999; Atkin and Morain, 1993). Methods using elliptic curves are derived from the planar geometric interpretation of elliptic equations, a general class of Diophantine equations, over finite fields (Davenport, 2008).

Definition 2.3.7. An elliptic curve, defined over the finite field $\mathbb{F}'_p$, has the form given below, where A and B are integers.

$$y^2 \equiv x^3 - Ax - B \pmod{p} \tag{2.3.2}$$

After Lenstra (1987) discovered that elliptic curves could be used for factorisation, the use of elliptic curves for primality testing was proposed by Goldwasser and Kilian (1986) and implemented as an algorithm called Elliptic Curve Primality Proving (ECPP) by Atkin and Morain (1993). It relies on a long known theorem regarding the bounds on elliptic-curve sizes to prove the primality of nearly every prime, or, conjecturally, all primes (Bernstein, 2004). ECPP is classified as a *Las Vegas* algorithm for the way in which the elliptic curves are switched until one which can be factored is found.

The best known version of this method uses elliptic curves with complex multiplications of small discriminants (Bernstein, 2007; Cheng, 2007; Lenstra and Lenstra, 1991) and is conjectured to run in heuristic random polynomial time. Bernstein (2004) puts the time complexity at the order stated below, however, Cheng (2007) puts it closer to $\tilde{O}(\log^6 n)$, or even $\tilde{O}(\log^3 n)$ when optimised using fast multiplication.

$$f(n) = O(\log^{4+o(1)} n) \tag{2.3.3}$$
$$\equiv O(\log^{4+\epsilon} n) \text{ for all } \epsilon > 0 \tag{2.3.4}$$

Furthermore, whereas Miller-Rabin provides a certificate of non-primality, ECPP provides a certificate of primality. The implications of this are that ECPP is extremely practical; it is fast, although slightly slower than Miller-Rabin, and has been used to prove the primality of a number $15,071$ decimal digits long (Brent, 2010). At present, AKS is insufficiently fast, and so, for large primes, the random time ECPP algorithm and error-prone Miller-Rabin algorithm are an acceptable compromise for applications where primes need be quickly certified.

2.4 The AKS Primality Test

The unconditional deterministic polynomial time algorithm from Agrawal, Kayal and Saxena (2004) is the first and only one of its kind. However, at present, it cannot compete with ECPP or Miller-Rabin on a practical level. This section provides an examination of the Agrawal-Kayal-Saxena (AKS) primality testing algorithm by outlining the work that has been done to improve it, as well as its implementations in practice over the years since it was developed. The reference to AKS being *embarrassingly parallel* in discussed and how this property lend itself to parallel processing on a high-performance computer. The reader is reminded of the notation $\log n$ to denote the binary logarithm of n ($\log_2 n$).

2.4.1 Overview of the Algorithm

Algorithm 2.4.1 The Agrawal-Kayal-Saxena (AKS) primality testing algorithm ($Z_n[x]$)

```
1:  procedure AKS($n$)
2:      if $n$ is a perfect power then
3:          return NOT PRIME;
4:      Find the smallest integer $r$ for which the order of $n \pmod r > \log^2 n$
5:      for each $a \le r$ do
6:          if $GCD(a, n) \ne 1$ then
7:              return NOT PRIME;
8:      if $n \le r$ then
9:          return NOT PRIME;
10:     for $a = 1, 2, \ldots, \lceil \sqrt{r} \log n \rceil$ do
11:         if $(x - a)^n \not\equiv x^n - a \pmod{n, x^r - 1}$ then
12:             return NOT PRIME;
13:     return PRIME;
```

The algorithm from Agrawal, Kayal and Saxena (2004) relies on the identity above, derived from corollaries of both Fermat's little theorem (once again) and another construct in combinatorics called *Pascal's triangle* (Aaronson, 2003). Proofs of this identity are also included in the works of Bronder (2006) and Menon (2013).

Lemma 2.4.1. The integer n is *PRIME* if, and only if, the following identity holds for all $a \in \{1, 2, \ldots, \lceil \sqrt{r} \log n \rceil\}$, where r is some integer such that the multiplicative order of $n \pmod r$, $\mathrm{ord}_r(n)$, is greater than $\log^2 n$ and $GCD(a, n) = 1$ for all such values of a:

$$(x + a)^n \equiv x^n + a \pmod{n,\ x^r - 1} \tag{2.4.1}$$

The final algorithm is constructed from this identity, with some additional conditions to ensure that no pseudoprimes are produced, i.e. no composite numbers incorrectly pass this test. The original time complexity was $\tilde{O}(\log^{12} n)$ but this has since been reduced to $\tilde{O}(\log^6 n)$ by using *Gaussian periods* (Lenstra and Pomerance, 2019) or by assuming the *Sophie Germain Conjecture* (Caldwell, 2021; Dietzfelbinger, 2004). The relative simplicity of AKS, and its use of concepts from Section 2.1, are evident in Algorithm 2.4.1.

2.4.2 Variants of the Algorithm

The original algorithm from Agrawal, Kayal and Saxena (2004) has undergone numerous iterations from both the authors themselves as well as others, most notably perhaps by Bernstein (2003), Lenstra and Pomerance (2019), and Crandall and Papadopoulos (2003). These works have served to reduced the complexity exponent by some half dozen orders of magnitude, mainly by reducing the polynomial operations in the congruence tests (line 11). This work aims to implement the variant below from Lenstra (2002), with a novel method from Crandall and Papadopoulos (2003).

Algorithm 2.4.2 The AKS variant of Lenstra (2002). Crandall and Papadopoulos (2003) use it to conceive the Z version.

```
1: procedure LENSTRA($n$)                                  ▷ Input: $n \in \mathbb{N}, n > 2$
2:     if $n$ is a prime power then
3:         return NOT PRIME;
4:     $r := 2$                  ▷ Finding such $r$ that order of $n \pmod r > \text{round}(\log^2 n)$
5:     while $r < n$ do
6:         if $GCD(r, n) \neq 1$ then
7:             return NOT PRIME;
8:         if $\text{ord}_r(n) > \lfloor \log^2 n \rfloor$ then
9:             break;
10:        $r := r + 1$
11:    for each $a \in [1, \phi(r) - 1]$ do
12:        if $(x - a)^n \not\equiv x^n - a \pmod{n, x^r - 1}$ then
13:            return NOT PRIME;
14:    return PRIME;
```

Before elaborating further, let us henceforth refer to implementations of AKS which use 'polynomials with coefficients modulo some number' as the $Z_n[x]$ version(s) to distinguish it from that proposed by Crandall and Papadopoulos (2003). Designated the Z version, their implementation method proposes the use of binary segmentation to multiply two polynomials to express them as a single large integer.

Doing this would effectively reduce the polynomial operation to one involving large-integer arithmetic. To this end, they hoped that this might allow for the exploitation of certain advantages, comprising a simplified implementation process, the use of arbitrary precision arithmetic and number theoretic software libraries, as well as faster arithmetic methods (i.e. FFT-based) for integer multiplication.

According to Crandall and Papadopoulos (2003), the time complexity of Lenstra's variant is given heuristically as $O \log^{6+\epsilon} n$, with runtime, T, empirically approximated as $T \sim C \log^6(n)$, where C is roughly constant over a range of n. This was achieved by changing the way r is obtained and by using the Euler totient function (Definition 2.1.6) to reduce the number of iterations needed for testing the congruence relation (Equation 2.4.1). Egan (2005) suggests that using the Carmichael function, instead of Euler's totient, can further reduce the runtime of this variant by a factor of four on average.

2.4.3 Implementation & Parallelisation

There are a plethora AKS implementations, parallel or otherwise, written in a multitude of languages. Serial implementations in C++ seem to be the popular choice and have been created by the likes of Li (2007) and Salembier and Southerington (2005), as well as four different ones by Gallot (2005). Those from from Gallot (2005) are among the best, three of which have arbitrary precision arithmetic and number theoretic software library dependencies and are addressed in Section 3.3, whilst the other is a basic standalone implementation without such optimisations. Further optimisation of the algorithm could come from the detection of perfect powers in linear time (Bernstein, 1998) as well as Karatsuba, which should be effective for RSA-sized numbers in the context of $Z_n[x]$, and faster (FFT-based) methods, in the case of the Z technique of Crandall and Papadopoulos (2003).

The AKS-class of algorithms, including both Z and $Z_n[x]$ versions, are considered 'embarrallel', or *embarrassingly parallel*, which means it can be separated into a number of tasks, which have little to no communication between them, with relative ease. In doing so, the algorithm can be implemented in parallel across multiple processors, or cores, to accelerate the computation. For AKS and its Lenstra variant, since the congruence tests in the loops in lines 10 and 11 of Algorithm 2.4.1 and Algorithm 2.4.2, respectively, are independent from one another and use predetermined values of a, the control flow is unaffected and, therefore, it is considered 'embarrassingly parallel', or embarrallel. Consequently, this stage of the algorithm can be split across many threads with minimal overheads and run on hundreds of GPU cores. This property, and the 'data-level parallelism' of the algorithm, is addressed at greater length, and in the appropriate detail, in Section 3.1.1.

A parallel implementation of AKS from Bronder (2006) favours the mathematics and the algorithm itself over pure parallel performance but showed that it could be done with a supporting library for faster computational arithmetic. Much has changed in the 15 years since the thesis of Bronder (2006), primarily in the pace at which computing hardware has advanced but also with respect to software libraries for computational arithmetic for large integers, which have been quietly and continuously developed and maintained in this time. These are the focus of the next section, the technology survey, and the results of a modern parallel implementation of AKS the targeted product of this research.

2.5 Summary

Despite barely scratching the surface, the introductions to number theory and complexity theory in this section is sufficient for understanding primes and the fundamental concepts related to them, as well as for analysing algorithms to approximate the running time and compare one algorithm to another. The study of primes segues into the exercise of primality testing, its significance to modern cryptography, and how centuries of effort searching for an algorithm which has deterministic polynomial time has engaged some of the finest minds in mathematics and computer science and led to the discovery of that from Agrawal, Kayal and Saxena (2004). Work will be conducted to investigate the Z version from Crandall and Papadopoulos (2003) which is believed to be novel, and yet to be implemented in practice.

Chapter 3

Technology Survey

In addition to substantial mathematical theory, this project incorporates a similar proportion of technology to practically implement the mathematics and the algorithms described in Chapter 2. As such, this is a survey of the relevant hardware and software technologies available for achieving such an implementation. The choice of programming language used to write the program containing the implementation of the algorithm is discusses, as are the supporting libraries available for optimising it and the GPU facilities on which it will ultimately be run.

3.1 Hardware & Implementation Facilities

Early implementations from the 2000s have used the likes of the 1.8 GHz *Pentium 4-M* processor (Salembier and Southerington, 2005), an 800 MHz *Pentium III*, a 1.42 GHz *Mac Mini* and the 1.0 GHz *Apple G4* machine (Crandall and Papadopoulos, 2003), as well as an eight node cluster of *Apple Xserve G5*s, each with two 2.0 GHz *PPC970* processors and 2 GB of RAM (Bronder, 2006), to prove the primality of integers in the hundreds of decimal digits in approximately one day.

By exploring the boundaries of today's parallel computing and GPU technology, this research seeks to understand just how fast an implementation can be made and whether such improvements can open it up to applications in the real world. In this section, an overview of parallel computing and a more comprehensive background in general-purpose computing on graphical processing units (GPUs) is given from an implementation perspective, as well as a description, and intended use, of the facilities available to achieve this.

3.1.1 Parallel and General-Purpose Computing on GPUs

Before proceeding any further, one should to stop and ask and consider whether or not parallel computing is a sensible solution to resort to. Bradford (2020) offers some guidance, strongly implying that parallel computing should be used as a last resort when it comes to solving a problem more quickly, and not before attempting both of the following (in order):

1. *Devising a better solution* - the potential for optimising the AKS algorithm has already been extensively explored mathematically by the likes of Lenstra and Pomerance (2019) (Section 2.4), as well as computationally, using libraries for arbitrary precision and number theoretic arithmetic, as discussed in Section 3.3.

2. *Getting a faster processor* - the modest personal facilities of the author already contains an eight-core AMD Ryzen 7 4700U CPU with a clock rate of up to to 4.10 GHz, which is not to be underestimated and could fare well for the smaller primes, perhaps up to 64 bit, but probably not up to the task of doing so for those in the 1024 or 2048 bit range.

Until such time that a better deterministic polynomial-time algorithm for proving primality is invented, another optimised revision of AKS is developed, faster processors are produced, or a faster software library for computational arithmetic is released, one may resort to parallel computing for this application. Furthermore, parallel computing has become far more accessible over the last 15 years. Multi-core processors are now commonplace in consumer electronics, and economic factors like increasing computation power-to-cost ratios for application specific devices mean that such devices are inexpensive but powerful (Harris, 2004). With this in mind, one can proceed with attempts to implement the algorithm in parallel.

Parallel computing, or parallel processing, is a type of computation whereby processes are carried out simultaneously across more than one computer processor, or core of a single multi-core processor. General-purpose computing on a GPU (GPGPU) is an emerging example of parallel processing and can also be carried out across multiple GPUs, expanding on the notion that computations on GPUs are already parallel in nature. In *CUDA*, the CPU (host) and the GPU (device) are linked to form a a heterogeneous system, where the CPU executes the serial program until it encounters a part of the code which is massively parallel. Since a CPU is not designed for parallel throughput, this is likely to cause a bottleneck. The parts of a program which can be parallelised are split into kernels and are launched (executed) concurrently as a set of threads, where each set of threads is called a grid, and each grid is mapped to a single *CUDA* core on the GPU.

The answer to why GPUs are suited to parallel computing, and application specific general-purpose computing, lies in their design from an architectural standpoint. GPUs are used for specialised, resource intensive computations whereas CPUs are designed for versatile serial computing for a range of applications. As such, dedicated GPUs are add-in circuit boards with a specialised processing unit on-board, as well as the additional infrastructure for system integration and fans and heat-sinks for cooling. Whereas a CPU might have a handful of cores, the processing unit on a modern high-end GPU can have tens-of-thousands of cores (NVIDIA, 2020; techpowerup.com, 2020) and an extremely high memory bandwidth.

Figure 3.1 compares CPU and GPU architectures and illustrates how, in comparison to a CPU, more transistors are dedicated to data processing than data caching and advanced control logic. The cores are able to work on thousands of threads simultaneously, making for a device which is specialised for compute-intensive highly parallel computation and delivers extreme computational power, measured in floating point operations per second (FLOPS).

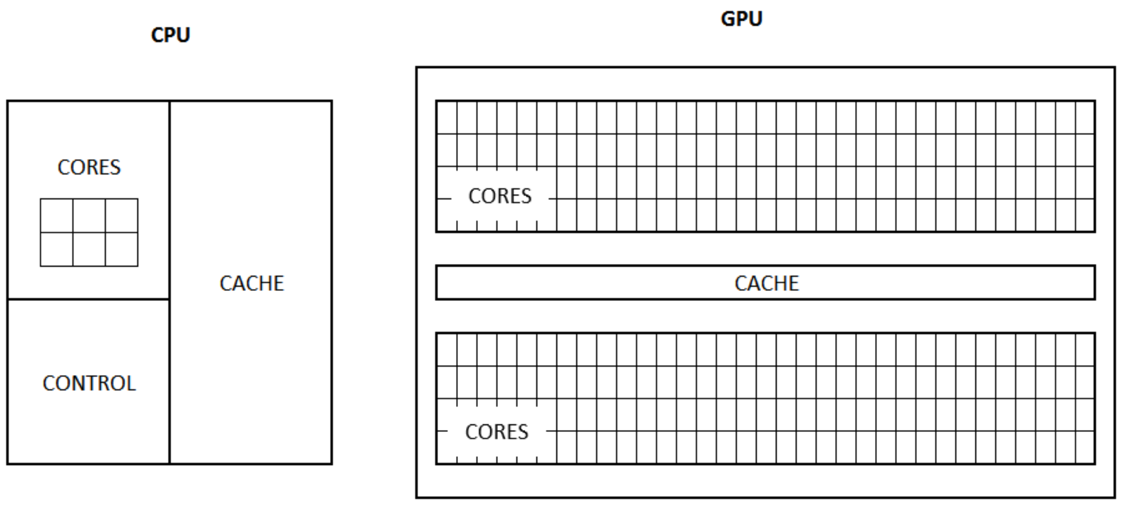


Figure 3.1: An oversimplified diagram comparing GPU and CPU hardware architectures. CPUs dedicate a majority of the silicon area to the cache and advanced control logic, whereas a GPU sacrifices memory access to dedicate the majority of its silicon area to a colossal number of cores, maximising computational throughput.

Recall that in Section 2.4.3, the AKS-class of primality testing algorithms are considered *embarrallel* (embarrassingly parallel), where, put simply, the problem can be separated into parallel tasks relatively effortlessly. Crandall and Papadopoulos (2003) describe this as analogous to C computers, all working independently on the same problem, solving it in t/C, where t is the time taken on a single computer. In addition to a constant control flow throughout the loop in line 11 of Algorithm 2.4.2, the congruence testing over a finite set of known values for a exhibits *data level parallelism*. This is where the same series of operations are repeated over a set of data, which may be exploited using a model of parallel computing known as *single instruction stream, multiple data stream* (SIMD).

Since every test on each value of $a \in [1, \phi(r) - 1]$ is unique and independent from another, each one may be computed in a separate processing thread. From Figure 3.2, one can discern how the domain $1 \leq a \leq \phi(r) - 1$ may be divided into approximately equal partitions corresponding to the number of cores available. Each core may then execute the same series of operations on every element in its domain, thus carrying an equal share of the computational load, solving the problem, in theory, in time t/c, where c is the number of processor cores available.

Core 1: $1 \leq a < \frac{1}{c}(\phi(r) - 1)$ Core 2: $\frac{1}{c}(\phi(r) - 1) \leq a < \frac{2}{c}(\phi(r) - 1)$

Core 3: $\frac{2}{c}(\phi(r) - 1) \leq a < \frac{3}{c}(\phi(r) - 1)$ $\ldots$ Core c: $\frac{c-1}{c}(\phi(r) - 1) \leq a \leq \phi(r) - 1$

This idea also gives rise to another execution model known as *single instruction, multiple threads* (SIMT), which takes the same concept of SIMD and applies it to *multi-threading*, where tasks are divided further across even more threads and each core is responsible for multiple threads. Parallel computing using a SIMT execution model is relevant to GPGPU since it allows a program to be implemented in parallel across multiple GPUs.

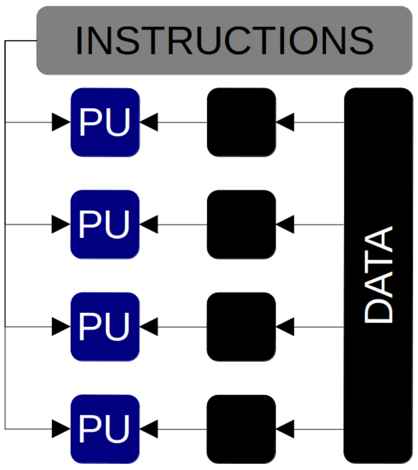


Figure 3.2: An oversimplified visualisation of the single instruction, multiple data (SIMD) parallel processing method used to exploit data level parallelism. Each core or processing unit (PU) within the multi-core processor takes on a partition of the data set, or domain of a for this application, and applies the same set of operations, i.e. the congruence relation. A quad-core processor is shown here, but, on a GPU, this would of course be scaled up enormously, to a core count numbering tens-of-thousands.

As well as being highly suited to data parallel computing, ease of programming an executable SIMD/T model is greater relative to other parallel models. Extending a program to implement a SIMD/T model may, nevertheless, still require extensive, hardware-specific refactoring. Herein lies the value of the *CUDA* API which aims to limit this burden by standardising the programming and deployment process on, in theory, any *NVIDIA* GPU. *CUDA* is discussed at length in the next section.

Applications of data parallelism can be limited and, therefore, algorithms which do not exhibit sufficient data parallelism would not benefit from the SIMD/T parallel model. Such algorithms might instead attain optimal performance using other parallel models, such as *multiple instruction streams, single data stream* (MISD) and *multiple instruction streams, multiple data streams* (MIMD) depending on the type of parallelism and control flow. Naturally, if there was no parallelism, there would be no benefit at all to computing in parallel, and one should instead focus on serial optimisation.

SIMD/T, and parallel computing generally, is not always a perfect solution, however, as coordinating the parallelism incurs overheads in form of additional computational cost. As such, the cost of a parallel computation is the sum of the computational cost of the process and that of management required to sustain the data parallelism, they must be balanced to ensure that these overheads do not become more costly than the task.

The vast computational power of modern GPUs has caused their general-purpose use to surge in commercial and research applications comprising deep learning, disease research, and proof-of-work based blockchain. This has, notoriously, had significant, far-reaching implications on energy use, raw material and semiconductor scarcity, as well as shortages and inflation in the GPU market itself due to a sharp soar in demand.

3.1.2 The *Compute Unified Device Architecture* (CUDA) API from NVIDIA

The *CUDA* toolkit from NVIDIA (2007) is a freely available development environment providing a parallel computing platform and programming model dedicated to general-purpose computing on all *NVIDIA*-made, *CUDA*-enabled GPUs. As well as comprehensive documentation for the hardware and the programming model, *NVIDIA* offers guidance in the form of dedicated support and training, offering seamless integration with hardware from the development environment, facilitating access to GPGPU by flattening the learning curve for programmers.

The fundamental purpose of *CUDA* is for programming tens-of-thousands of cores on a GPU efficiently and use them in combination with a CPU as a heterogeneous system to exploit the advantages of both kinds of processor to execute a particular task in parallel. Programming is similar to C/C++, with some syntactic additions for parallel execution, and is supported through a dedicated compiler but, there is also third-party support for languages including, but not limited to *Haskell*, *Python*, and *MATLAB* (NVIDIA, 2021a).

CUDA supports 64-bit integers but operations on these types compile to multiple instruction sequences on some GPUs depending on their general specifications and available features (NVIDIA, 2021c). Interestingly, *CUDA* also includes its own Fast Fourier Transform library (NVIDIA, 2021b), the capabilities of which could be explored or, at the very least, offer a certain level of redundancy should the author encounter any issues integrating the selected third-party library. Furthermore, in addition to simply offering accelerated computing on a single GPU, *CUDA* offers as a high level of parallelism by allowing work to be distributed across multiple GPUs (NVIDIA, 2021c), yielding performance benefits, in terms of certification time, which could also be investigated.

Finally, *CUDA* C/C++ and the associated compiler provides an abstraction by producing code that is not hardware-specific, meaning that, once written and working, *CUDA Kernels* will work on any *NVIDIA* or, more generally, *CUDA*-supported GPU architecture (NVIDIA, 2021c). Needless to say this is a useful implication, the benefits of which are self evident: any implementation of the AKS primality test in *CUDA* can be implemented on next-generation hardware, as and when it becomes available, without any refactoring, save for further optimisation of the algorithm itself using new libraries, arithmetic algorithms, or parallel processing methods.

3.1.3 The *Hex* GPU Cloud

Hex is the name given to the high performance computing facility belonging to the Department of Computer Science at the University of Bath. The *Hex* is a seven-node GPU cloud computing facility, remotely accessible to members of the department using the secure shell protocol (SSH) (Haines, 2021). Using the *OpenSHH Client* on *Windows*, institutional credentials, and the domain name server (DNS) entry of the chosen node, the author can access the facilities from any of their personal machines. The GPU facility offers a diverse range of hardware specifications to choose from, depending on which node is selected. Evidently, selection of the appropriate hardware required is dependent on the application.

Given the pace of development in hardware and general-purpose computing on GPUs, any performance data produced will inevitably become antiquated after a number of years. Subsequently, to prolong the applicability of the data and prevent this from happening to an extent, the author proposes that the implementations be carried out on the latest available hardware which, at the time of writing, is the *NVIDIA GeForce RTX 3090* multi-core GPU.

Launched in September 2020, the chipset on this device consists of a single 1395 MHz base-clock GPU comprising of 82 streaming multiprocessors (SMs), making for a total of 10,496 *CUDA* cores in addition to 328 *Tensor* and 82 *Ray Tracing* (RT) cores. NVIDIA (2020) claim that, through their proprietary *Ampere* architecture, both data paths on each streaming multiprocessor can be used for single-precision (FP32) floating point operations, effectively doubling the core count and the raw processing throughput of the GPU to 35.6 Tera-FLOPS (floating-point operations per second). This is paired with 24 GB of 1219 MHz memory on a 384-bit memory interface bus, producing a memory bandwidth of 936 GB - nearly a terabyte - per second (techpowerup.com, 2020). While the floating point operations statistic, stated here, is a standard benchmark for evaluating GPU performance, one more pertinent to the context of primality testing is that relating to the 32-bit integer operations (INT32) stated as 17.8 Tera-OPS (operations per second) on 5248 cores (Walton, 2020).

Since February 2021, the *Hex* has eight of these flagship graphics cards installed, four of which are available on one particular node and being theoretically available from June of the same year. Exactly how the raw computing power of this hardware can be exploited effectively will be one of the implementation challenges and should become clearer with time and experience. While other, adequately preforming hardware is also available on other nodes, selection of this particular model of hardware on the node named *nitt* (etymology unknown) is the pinnacle of the current generation of high-performance computing, thereby prolonging the ‘shelf life’ and, consequently, the relevance of the data should it be reused for any further work or comparisons in future.

With regards to compatibility between firmware and the intended hardware, given that the intended parallel computing API, *CUDA*, for implementing the refined algorithm in parallel and the target hardware both originate from the vendor *NVIDIA*, there is dedicated support and documentation for these resources meaning that cross-compatibility is not expected be an issue. However, should this not prove to be the case, any difficulties should be relatively straightforward to resolve for the same reasons. Issues with this implementation are more likely to originate from interfacing with the *Hex* cloud itself, or from the installation and use of the supporting libraries.

Whilst, nodes on the *Hex* have a number of *Python 3* libraries as well as development, data, and processing tools included, the list does not include any of those described in Section 3.3. Consequently, inclusion of the relevant libraries necessary for large integer arithmetic and faster multiplication algorithms will have to be achieved accordingly using *Docker-via-Hare* which is supported by the *Hex*. Furthermore, the *bash* shell for the *Hex* command line is scriptable, enabling automation of any processes, laborious or not, involved with implementing the final program in parallel. Full details of any library installations, bash scripting, as well as compilation and parallel processing of the implementation, are documented in Chapter 4 and Chapter 5.

3.2 Programming Languages

The AKS algorithm could potentially be written as a program in any language, however, with the primary aims of parallel implementation, speed and time savings in mind, the choice is narrowed to a set of viable compiled languages able to deliver this adequately at run-time. The chosen language for the implementations will be C++, a general-purpose programming language with imperative, generic, object-oriented, as well as functional features, and indeed the language of choice for Salembier and Southerington (2005), Bronder (2006), and Li (2007).

As a mainstream programming language, it benefits from an abundance of supporting libraries, many of which facilitate the inclusion of faster arithmetic and number theoretic methods crucial to the improving computational time complexity of the implemented algorithm. Salembier and Southerington (2005), Bronder (2006), and Li (2007) were all able to optimise their programs using supporting number theory libraries for C++, such as NTL (Section 3.3.2) and GMP (Section 3.3.1). Being an important factor in the selection of the language, the relevant libraries available for this application are listed and discussed in greater detail in the next section (Section 3.3).

A second property of C++, key to its selection, is the close resemblance it bears to that of the *CUDA* API. The use of *CUDA* to develop high-performance, GPU-accelerated algorithms and facilitate the implementation of them on hardware from *NVIDIA* is discussed at length in sections Section 3.1.2 and Section 3.1.3, the main idea here being that portability of the program from a basic, non-parallel CPU-based to an accelerated one in parallel on a GPU is simplified. Similarity of the languages will avoid unnecessary translation of code or integration of third-party wrappers to implement unsupported languages, consequently saving time without compromising performance.

An honourable mention should be given to the purely functional programming language, *Haskell*, whose mathematical integer properties and potent supporting number theory libraries, such as *arithmoi* from Lelechenko and Fischer (2020), make it a serious contender to C++ as an implementation language for a standalone program. However, one is not simply delivering a standalone program but rather one running in parallel and, therefore, efforts will be focused on developing a C++ implementation due to precisely these circumstances and the overwhelming advantages this language has in favour of them. Nevertheless, an implementation of the AKS algorithm in Haskell could serve as an interesting and informative exercise.

3.3 Supporting Libraries

The C++ programming language, like many others, benefits from a multitude of third-party libraries, supporting a plethora of applications by including additional functionality. The value of libraries lies in the optimisation they can offer application-specific tasks. For efficient manipulation of larger integers, as is the case here, libraries use the algorithms covered in Section 2.2 to condense the number of computational steps involved in an arbitrary mathematical operation, consequently reducing the time complexity.

Even small, individual gains can be accrued for operations repeated a large number of times, yielding significant improvements overall. Such gains are measured, theoretically, in terms of time complexity, which translates, in practice, to the total length of time taken for the algorithm to run to completion. A non-exhaustive selection of relevant libraries, and their capabilities, is listed and assessed against a set of criteria including, but not limited to, ease of use with C++, ease of use with *NVIDIA CUDA*, range of features, performance, and mathematical methods.

3.3.1 GMP - The GNU Multiple Precision Arithmetic Library

The GNU Multiple Precision Arithmetic Library (GMP) from (Granlund, 1991) is a freely available, versatile, feature-rich library for arbitrary precision arithmetic. GMP was the chosen number theory library for the parallel and non-parallel implementations of Rotella (2005) and Bronder (2006) respectively. It was also included in the first of three non-parallel, big number dependent implementations from Gallot (2005).

From the discussion around almost all of the other libraries in this section, the influence and versatility of the GMP library is already plain to see. It almost always paired with one of them as the underlying multi-precision integer library for given implementation suggesting that there are numerous advantages to doing so. Such advantages related to pairing it with NTL specifically are outlined in the next section (Section 3.3.2). The GMP library enables probabilistic primality testing (Miller-Rabin or Baillie-PSW, see Section 2.3), integer addition, subtraction, multiplication, division, greatest common divisor (Euclidean), modulo, exponentiation, and modular exponentiation, the implementations of which are described in the GMP documentation (Granlund, 2020). For multiplication, either one of Karatsuba, fast Fourier transforms, or Toom-Cook multiplication, is automatically chosen depending on the size of the number.

While GMP is described as high-performing, it would appear to be among the slower libraries from a computational number theoretic standpoint. Of the three library accelerated implementations of AKS from Gallot (2005), the one including GMP exclusively is the slowest of them all. It should be noted, however, that this is *relative* to his NTL- and MIRACL-supported (Section 3.3.3) implementations and, therefore, the GMP one is still probably very fast in its own right. The thesis of Bronder (2006) also acknowledges that GMP is not the fastest arbitrary precision arithmetic library available, however, it was, as he describes himself, perfectly suited to his work, 'focusing on the algorithm and not the speed'. The focus of *this* work is, in a sense, the opposite.

3.3.2 NTL - A Library for doing Number Theory

The number theory library (NTL) by Victor Shoup (2001b) is a highly reputable C++ library and was the preferred choice of Salembier and Southerington (2005) and Li (2007) for their non-parallel programs. Furthermore, it was another of the libraries used to apply fast Fourier transforms for polynomial multiplication in the trio of accelerated implementations from Gallot (2005), though not the fastest of the three. The NTL library features arbitrary length integer arithmetic as well as thread safe and multi-core capabilities to accelerate low-level computations. It is freely available under version 2.1, or later, of the GNU Lesser General Public License.

To obtain optimum performance from NTL and for facilitating installation, Shoup (2001b) recommends *Unix* or *Unix*-like (*Linux*) platforms which is pertinent to this project given that the parallel implementation facility, the *Hex* (Section 3.1.3), would be considered a *Unix*-like environment. Any opportunity to gain time- or performance-related benefits should be exploited, and this qualify as one such opportunity. Another opportunity is the tenfold or more performance enhancements it supposedly gains from building it in a *Unix*, or *Unix*-like, environment and in conjunction with GMP, the multi-precision arithmetic library from the last section (Section 3.3.1).

Furthermore, a relatively recent addition to NTL is the *thread boosting* feature which uses available threads in a thread pool to accelerate computations on multi-core machines. Whether this feature applies to large integer arithmetic and algorithms in NTL remains to be understood, however, both the *Unix* and thread-boosting options, although potentially more advanced, should be considered when including NTL in the implementation.

According to the library documentation (Shoup, 2001a), the module specifically relevant to this application is the class enabling arbitrary length integers and includes routines for greatest common divisors, modular arithmetic, primality testing and small prime generation. Asymptotically fast algorithms are implemented for long integer multiplication, using the classical algorithm or Karatsuba for very big numbers, and long integer division, using only the classical algorithm unless NTL is used in combination with GMP in which case it employs the GMP algorithm.

For polynomial multiplication and division, either the classical algorithm, Karatsuba, fast Fourier transforms using small primes, or the Schönhage-Strassen approach to fast Fourier transforms is automatically chosen depending on the domain of the coefficients.

3.3.3 MIRACL - Multi-precision Integer and Rational Arithmetic C/C++ Library

In the last section, it was said that the fastest program in the set of Gallot (2005) did not use NTL. That honour goes to the one supported using the Multi-precision Integer and Rational Arithmetic C/C++ Library (MIRACL) by Scott (n.d.). This library is claimed by its developers as being the 'gold standard' open source software development kit for elliptic curve cryptography and boasts commercial adoption among an elite clientele in a range of development environments such as embedded, SCADA, and mobile.

Essentially, it is a C library which provides an inline C++ wrapper and C++ interface as well as hardware optimisation, cryptographic, and number theoretic capabilities.

Of these, C++ compatibility and provision of number theoretic techniques is of obvious convenience but universally prevalent among the other libraries considered here. However, the possibilities for optimization of both embedded processors and RAM to help overcome device and memory constraints are noteworthy since MIRACL is unique in this regard. If these constraints are identified as serious limitations during the parallel implementation phase, the potential of this particular feature could, and perhaps should, be explored further.

In the implementation from Gallot (2005), the polynomial multiplication capabilities of the library are leveraged to produce, supposedly, the fastest of his three library-accelerated C++ implementations. The library includes big number formats and has a selection of low-level, advanced, and modular routines at its disposal. The low-level routines includes operations on big integers comprising of addition, subtraction, division, multiplication, modulo, divisibility checks, and greatest common divisors (Euclidean). Advanced routines consist of the likes of random big number generators, a probabilistic primality test (exact kind unknown), prime and safe prime number generators, Chinese remainder theorem, and fast Fourier transforms, with a method for finding or verifying perfect powers notably absent.

The precise details of each method and the way in which they are used are detailed in full in the MIRACL reference manual and therefore beyond the scope of this survey, meaning further elaboration beyond some noteworthy or relevant routines, and the methods behind them, outlined above is unnecessary. It is evident from descriptions given in the last paragraph that MIRACL is potent and highly capable library contributing multiple highly relevant methods and subroutines for optimising an AKS implementation in C or C++. As it currently stands, including MIRACL as the primary supporting library and supplementing it with methods from NTL would produce a substantially optimised standalone implementation of algorithm to implement in *CUDA*, subject to compatibility and facility of integration of course.

Given that MIRACL has commercial links to a company named after the software library itself, *MIRACL UK Ltd.*, licensing and copyright must be addressed for the sake of diligence. The rights to the library are reserved by this company, who make it freely available to redistribute and/or modify under the terms of the *GNU Affero General Public License*, version 3 or later, as published by the *Free Software Foundation*. The scope of the project is bound, at most and in an ideal scenario, to a proof of concept in an academic context. As such, it is not intended to be used to develop commercial activities involving MIRACL or to ship MIRACL with a closed source product, meaning it is within the requirements of the license and, therefore, purchasing a commercial license will not be necessary.

3.4 *CUDA* and Unsupported Libraries

It is generally not possible to use a library written for CPUs to be used in a GPU kernel, at least, not efficiently. This is down the differences in architecture which mean that a GPU works very differently from a CPU all the way from the execution model up to the actual instructions and the binary code. In the case of GMP, the library is compiled for x86, i.e. host only, and doesn't include functions compiled for use on the device.

The unfortunate reality of this is that none of the libraries mentioned here, including the NTL and GMP libraries, will compatible with *CUDA*, and it would appear that it would take a lot of work to redesign them to run on a GPU. Worse still, the built in *CUDA* library for computing discrete Fourier Transforms (NVIDIA, 2021b), *cuFFT*, does not have the necessary functionality for performing computational number theory as effectively as NTL, nor do any of the other *CUDA* libraries, such as *cuBLAS* or *cuSPARSE* and *cuRAND*, it would it seem. In terms of other alternatives, there are archived web pages online for *The CUDA Multiple Precision Arithmetic Library*, or *CUMP*, however the most recent releases for this are from the year 2012 which would suggest development is discontinued. This seems to be the case for another alternative, this time found on *GitHub*, named the *CUDA accelerated(X) Multi-Precision library*, or *XMP*, and filed under *NVIDIA Research Projects*.

In academic literature, the works of Ewart, Hehn and Troyer (2013) and Joldes et al. (2016) appear to be a couple of the most recent and most pertinent to this application, however source code for these appears to be unavailable and their current state of development unknown. The search for a supported *CUDA*-compatible library for arbitrary precision integer arithmetic has proven to be fruitless but not without promise; academic papers on the subject have measured speed-ups on decade-old (at the time of writing) GPUs many tens of times faster than a comparable operation on CPUs using the NTL and GMP libraries. Given the rise in notoriety of GPGPU and its transformative applications in artificial intelligence and blockchain technology, it may not be too much of a stretch to speculate that within a few years there will be a range of libraries for computational arithmetic on GPUs, much like there is today for single thread multiple precision arithmetic on CPUs. Such an undertaking could be a PhD project in itself or a task for the open source community, from which one may catch on and gain a reputation in a similar way to NTL and GMP have done in their own domains.

Until such time, avenues for algorithmic optimisation, parallel strategies and implementations of AKS can still be explored and evaluated to a certain extent by running them in parallel on multi-core CPUs. In doing so, more light may be shed on the problem in terms of speeding up the algorithm itself or gaining experience and knowledge in preparation for a time when deployment on a GPU is more straightforward than it currently is.

3.5 Summary

This review has provided the necessary insights into computational arithmetic and number theoretic software libraries, GPU architecture, data parallelism, and *CUDA* to work towards implementing AKS in parallel on a small scale. The direction concerning the implementation of AKS in parallel is, in theory, unambiguous; a successful implementation of AKS seeks the practicality of Miller-Rabin and ECPP, in practice, whilst retaining deterministic behaviour and optimal polynomial time complexity. In practice, this is likely to be difficult without any GPU- or *CUDA*-compatible computational number theoretic software libraries, necessary for the arbitrary length integer arithmetic that will allow GPGPU to be applied to AKS *and* be practical for testing very large inputs. For applications such as this, there is a demand for a highly competent GPU library for arbitrary precision arithmetic similar to that which exists for CPUs with NTL and GMP.

Chapter 4

Research Projections & Methods

Now informed by a thorough review of the relevant literature and a survey of the compatible technology, the research objectives and deliverables have been revised accordingly.

4.1 Objectives

Three broad research objectives have been identified and are described as follows:

1. Develop the Z version of the algorithm which uses binary segmentation as the primary technique for polynomial multiplication and evaluate the performance of this variant relative to an implementation of the $Z_n[x]$ version using other asymptotics for polynomial multiplication, such as Karatsuba, FFT, and CRT.
2. Explore the application of GPGPU on AKS, and its use in primality testing generally, by deploying AKS programs in parallel on a modern, multi-core GPU (preferably the latest available, which is the *NVIDIA GeForce RTX 3090* at the time of writing). Explore the potential benefits, if any, of implementing the algorithm across multiple GPUs.
3. Determine whether certification of large primes is possible using both variants of the AKS algorithms and, if so, make a judgement as to whether the method is at all practical, particularly against randomised algorithms.

4.2 Deliverables

4.2.1 Desired Outcomes

The points above can be broken down further into deliverable actions which, if completed, will ultimately lead to meeting all objectives. The deliverables are listed below and are considered a complete list of ideal outcomes should project go entirely to plan.

1. Develop a program which correctly performs multiplication operations on univariate polynomials with integer coefficients using binary segmentation. Optimise this for time complexity and run-time performance.
2. Evaluate the performance and viability of binary segmentation multiplication and compare it with other prominent methods, such as Karatsuba, FFT, and CRT.
3. Successfully implement the Z variant of the Agrawal, Kayal and Saxena (2004) primality testing algorithm, using binary segmentation algorithm as the primary technique for polynomial multiplication operations.
4. Evaluate the performance of the Z variant by comparing with an identical implementation which uses the other asymptotics for polynomial multiplication.
5. Demonstrate competency and familiarity with parallel computing concepts, the parallel computing API, *CUDA*, and the operation of the University of Bath GPU cloud facility.
6. Use the parallel computing API, *CUDA*, to port the best performing program of the two AKS algorithm variants for deployment available on a modern, multi-core GPU (preferably the latest available, which is the *NVIDIA GeForce RTX 3090* at the time of writing) using the *Hex* GPU cloud facility at the University of Bath.
7. Produce experimental data in the form of parallel processing performance benchmarks for modern hardware. Explore the possibility and potential benefits, if any, of implementing the algorithm across multiple GPUs.
8. Measure the performance of the algorithm against existing methods such as Elliptic Curve Primality Proving (EPCC), the Miller-Rabin Primality Test, or a combination of both. Quantify the above and produce a cost-benefit comparison between each.

4.2.2 Alternative Endpoints

In the event of any deviation from the original plan due to intentional changes of direction, unforeseen circumstances, time constraints, or other difficulties, any of the alternative criteria below could still be considered a successful outcome if they are met. The enumeration of these indicates the degree of success relative to the ideal outcomes above, i.e. 1 achieving closest to the ideal outcomes without fulfilling them all, 2 being another level short of objectives than 1, and so on. These are defined as follows:

1. An AKS primality algorithm, optimised using an appropriate library to include fast arithmetic methods, is successfully implemented in parallel on the Bath Hex GPU cloud. However, performance data for benchmarking purposes is not gathered or insufficient for making effective comparisons with other algorithms. Consequently, the results, if any, may only be treated as indicative given that concrete conclusions cannot be drawn directly.

2. Both Z and $Z_n[x]$ variants of the algorithm are implemented correctly, complete with performance profiling for comparisons to be made between the two programs their corresponding polynomial multiplication algorithms. Both programs are in a state such that they may be ported across to *CUDA* for deployment on a GPU relatively easily. Exploratory, but ultimately unsuccessful, attempts are made at implementing the best performing of the two versions in parallel on a GPU. An account of this exploratory work and recommendations for future work in this vein should accompany the preliminary algorithmic and development work.

3. The Z version of the algorithm is implemented successfully using a. Comprehensive performance profiling of this polynomial multiplication method has been completed and some comparisons have been made with an implementation of the $Z_n[x]$ version.

The three cases described above are the minimum necessary requirements which must be satisfied for the project to be deemed a success to some degree. Within each of these, success can also be assessed more generally, to a further degree, according to the criteria below. Compromises made on any of these, should be documented with an appropriate justification.

- The supporting library for faster arithmetic is the fastest, preferred, or most compatible choice from the libraries available.

- The implemented variant of the AKS algorithm is the fastest, or preferred, or most suitable choice from the variants available.

- The extent of optimisation of the chosen AKS variant, either at the algorithmic or implementation (code) level.

4.3 Methods

4.3.1 Development Environment and Software Libraries for Computational Arithmetic

Version 11.5.1 of NTL was compiled from source and installed in tandem with GMP, version 6.2.0, to exploit every opportunity to accelerate performance, including the ten-fold or more performance gains potentially available with a setup such as this. This development environment falls in line with the recommendations in the NTL documentation, which are summarised in Section 3.3.2. Installation of MIRACL was attempted but installation using instructions given was both ambiguous and unsuccessful so attempts to include it were abandoned. Late on in the development of this project, another library called FLINT and developed by Harvey and Hart (2007) was discovered. Being very similar to NTL in many respects, as well as comparable in performance, attempts to install and experiment with this library were also attempted but its was never included in the project in any form.

4.3.2 Process

An implementation of the $Z_n[x]$ version of Lenstra's algorithm has been optimised for performance, using techniques listed in Section 5.2.5. This version uses the fastest computational asymptotics available in the NTL library to correctly implement this version of AKS in serial. From here, this implementation is forked and an almost identical program is developed on this branch and the Z version. Congruence testing is included for the Z and $Z_n[x]$ as a separate function header in order to more easily switch between running the two and avoid conflicts. These congruence testing functions are named `CongruenceZnx` and `CongruenceZ` appropriately to avoid any conflicts.

Per the recommendations of Crandall and Papadopoulos (2003), binary segmentation will be implemented and the primary polynomial multiplication algorithm and used within a bespoke *powermod* function. The development work involved in this is detailed in full in Section 5.2.1. The $Z_n[x]$ implementation will serve as a benchmark against which the performance of the newly developed Z version will be compared and reported in Section 5.2.3. For the Z version the new runtime is anticipated to be proportional to r and the overheads incurred from switching between polynomials and integers when going from the $Z_n[x]$ and Z versions respectively are expected to be between a factor of two and four.

4.3.3 Test Primes and Polynomials

A set of primes, ranging in length were compiled from multiple sources, such as Li (2007), Egan (2005), Bronder (2006), and Gallot (2005). This list of primes included in Appendix B. To further enhance this set of test specimens, *Maple* software was used to produce subsets of primes in the range $p \in \{i \mid N - 210 \leq i \leq N\}$ from the original seed primes, N. Domains of this size across varying magnitudes of N enables the investigation of recurring patterns across a constant range. This enhanced list of primes in too long to include in this document but is made available in the source code repository.

4.3.4 Computing Resources

Due to the lack of `sudo` privileges on the `linux.bath.ac.uk` facility, the set of computational resources available were somewhat limited to a range of personal machines comprising two desktop computers and a laptop. Hardware specifications varies between each of these, providing a useful spectrum of computing power to develop the programs and test their performance in practice. The relevant specifications are found in full in Section A.1. Given the impossibility of any NTL- or GMP-enabled programs, implementations of the AKS algorithm can be deployed in parallel on the multi-core CPUs within these devices thanks to the thread pool functionality of the NTL library. This is sufficient for the purposes of this project now that the focus has shifted more towards the development and evaluation of the Z version of Lenstra's AKS variant.

4.3.5 Measurement and Quantification

All program Execution and function run-times are measured in milliseconds using the `high_resolution_clock` from the standard `chrono` C++ library. If this technique has been unable to resolve the time between instructions (the language is known for being especially fast at run-time generally), then parts of the program have been set to loop enough times for a mean to be calculated or for the time to be resolved and then compared over a number of runs rather than over a single one.

The reliability of this clock from the inbuilt libraries is often said to be questionable and dependent on running temperature (among other factors), but, for serial computing, the CPU load is often balanced sufficiently between cores to maintain optimal performance and this, is perfectly acceptable. Problems may arise, however, when running programs in parallel and the performance of the CPU may be choked or bottle-necked deliberately to protect it from overheating. In such cases, any timed parallel computations will be run over a number of cores less than the maximum number available allow the load to be managed appropriately and reduce the chances of this occurring.

Chapter 5

Implementation & Optimisation

This chapter details the major refactoring and program development undertaken to both optimise existing work and extend it, respectively, to deliver the project objectives outlined in Section 4.2. This comprises the development of novel functions to introduce new algorithms, the implementation of the necessary functionality and infrastructure required to compile and run the programs in parallel, as well as minor optimisation techniques to the base code generally.

5.1 Binary Segmentation

5.1.1 In Theory

In mathematics there exists a corollary that polynomial multiplication is equivalent to acyclic convolution. Consequently, one can manipulate two polynomials to construct an integer signal with strategically placed digits corresponding to the coefficients of their polynomial product, from which they may then be inferred. This procedure, formally known as *binary segmentation*, has been adapted from Crandall and Pomerance (2006) to accommodate the occurrence of signed, or negative, coefficients and is given below.

Algorithm 5.1.1 The binary segmentation algorithm for multiplying two polynomials, $f(x) = \sum_{j=0}^{U-1} f_j x^j$ and $g(x) = \sum_{k=0}^{V-1} g_k x^k$, where f_j & g_k are real integer coefficients, U & V are the number of terms in each polynomial, and the product $s(x) = f(x) \cdot g(x)$ is given in signal form. Reproduced from Crandall and Pomerance (2006).

```
1: procedure MULTIPLY($f(x)$, $g(x)$)    ▷ Input: Two polynomials, $f(x)$ and $g(x)$.
2:     Choose $b$ such that $2^b > \max\{U, V\} \cdot \max\{f_j\} \cdot \max\{g_k\}$    ▷ Initialise.
3:     $F = f(2^b)$    ▷ Create binary segmentation integers.
4:     $G = g(2^b)$
5:     $m = F \cdot G$    ▷ Integer multiplication.
6:     for each $i \in \{n \mid 0 \le n < U + V - 1\}$ do    ▷ Reassemble coefficients into signal.
7:         $s_i = \lfloor m/2^{bi} \rfloor \mod 2^b$    ▷ Extract next $b$ bits.
8:     return $s(x) = \sum_{i=0}^{U+V-2} s_i x^i$    ▷ Base-$b$ digits of $m$ are desired coefficients.
```

To demonstrate the effectiveness of this method, two polynomials of degree 2, below, shall be multiplied using decimal segmentation (since binary is not being used here).

Let:

$$f(x) = x^2 + 2x + 1 \tag{5.1.1}$$
$$g(x) = x^2 - x + 1 \tag{5.1.2}$$

Evaluating the polynomial product of the two traditionally give the following:

$$\begin{aligned} s(x) &= f(x) \cdot g(x) \\ &= (x^2 + 2x + 1) \cdot (x^2 - x + 1) \\ &= x^4 + x^3 + x + 1 \end{aligned}$$

Decimal segmentation is comparable to the binary equivalent and the result of the polynomial multiply can be obtained using Algorithm 5.1.1, with minor modifications:

$10^b > \max\{D, E\} \cdot \max\{f_j\} \cdot \max\{g_k\}$

$10^b > \max\{2, 2\} \cdot \max\{2\} \cdot \max\{1\}$

$10^b > 2 \times 2 \times 1$

$10^b > 4 \therefore$ for non-negative polynomials $b > 0$, so let $b = 1$.

$10^b = 10^1 = 10$

$$\begin{aligned} F &= f(10^b) & G &= g(10^b) \\ F &= f(10^1) & G &= g(10^1) \\ F &= (10)^2 + 2(10) + 1 & G &= (10)^2 - (10) + 1 \\ F &= 121 & G &= 91 \end{aligned}$$

Comparing the output from the decimal segmentation procedure, by parsing the integer in Equation 5.1.4, with that from doing the multiplication 'by hand' Equation 5.1.3, one can observe the correlation between the placement of the digits and the coefficients of the polynomial product. In practice, the solution will probably never be as trivial as that in this example, however, this is sufficient for demonstrating the concept.

$$\begin{aligned} s(x) &= f(x) \cdot g(x) \\ &= x^4 + x^3 + x + 1 \qquad (5.1.3) \\ (&\equiv x^4 + x^3 + 0x^2 + x + 1) \end{aligned}$$

$$\begin{aligned} u &= F \cdot G \\ &= 121 \times 91 \\ &= 11011 \qquad (5.1.4) \end{aligned}$$

Crandall and Pomerance (2006) report the bit-complexity of multiplying two degree-D polynomials, with all coefficients reduced modulo n, as $O(M(D \ln(Dn^2)))$, where $M(b)$ is the relevant bit-complexity of multiplying two integers, each of b bits. With an efficient integer multiplication algorithm, this method of computing the product of two polynomials could reduce laborious and repeated substitution and reduction procedures to a single high-precision integer multiplication.

The same could also be said for a congruence test of two polynomials, such as that of Equation 2.4.1 implemented in line 12 of Algorithm 2.4.2. For the congruence testing routine, binary segmentation could replace the many polynomial operations with a single multiplication to optimise complexity and running time. Considering that Equation 2.4.1 involves $(r-1)$-degree polynomials and typically have coefficients of size n (Crandall and Papadopoulos, 2003), one can envisage the relative simplicity of generating the large integer m and asserting the equivalence of the two integers (i.e. whether the relation holds).

5.1.2 In Practice

Implementing binary segmentation required several modifications to be made to the original algorithm to allow it to work correctly and be applied practically in the context of this application. Among the first of these was a new modulus; initialising using $2^b - 1$ provides additional padding to accommodate signed coefficients. After that, any subsequent cases of cell overflow in the signal are handled by new the conditions set out after the extraction of each coefficient.

Algorithm 5.1.2 A variant of the binary segmentation algorithm, adapted from that given by Crandall and Pomerance (2006) to accommodate signed, or negative, coefficients. Notation and symbolic representation is unchanged from Algorithm 5.1.1. Code implementing this algorithm in C++ may be found in Appendix C.

```
1: procedure MULTIPLY($f(x)$, $g(x)$)          ▷ Input: Two polynomials, $f(x)$ and $g(x)$.
2:     Choose $b$ such that $2^b - 1 > \max\{U, V\} \cdot \max\{f_j\} \cdot \max\{g_k\}$     ▷ Initialise.
3:     $F = f(2^b - 1)$                          ▷ Create binary segmentation integers.
4:     $G = g(2^b - 1)$
5:     $m = F \cdot G$                             ▷ Integer multiplication.
6:     for each $i \in \{n \mid 0 \leq n < U + V - 1\}$ do   ▷ Reassemble coefficients into signal.
7:         $s_i = \lfloor m/(2^b - 1)^i \rfloor \mod 2^b - 1$           ▷ Extract next $b$ bits.
8:         if $s_i > \frac{1}{2}(2^b - 1)$ then
9:             $s_i = s_i - (2^b - 1)$
10:            $m = m + (2^b - 1)$
11:    return $s(x) = \sum_{i=0}^{U+V-2} s_i x^i$       ▷ Base-$b$ digits of $m$ are desired coefficients.
```

A C++ program implementing this new algorithm (above) may be found in Listing C.5, as well as another variant, discussed later, containing further adaptations for multiplying polynomials with coefficients modulo some integer h. For these, time complexity is estimated to be $O(D_s)$, where D_s is the degree of the output: the polynomial product, $s(x)$. Testing of the program would suggest that this complexity is at the higher end of linear-time since it is slow in comparison to the other NTL polynomial multiplication methods.

Full performance profiling of the program in the form depicted in Algorithm 5.1.2 is given at length in the next section, however areas for improvement can already be identified. The complexity generally suffers where where bespoke helper functions have had to be made to add certain functionality absent from the NTL library. These include functions for the substitution steps in lines 3 and 4, as well as for obtaining the greatest coefficients, $\max\{f_j\}$ and $\max\{g_k\}$.

Whilst some efforts were successful at improving the runtime of these subroutines by as much as 65% for polynomials of small degree, they are unlikely to see significant improvements until they are given the appropriate attention, or even the NTL treatment and feature as dedicated components of the library. Aside from benefiting from new library functionality, more could perhaps be made of bit-wise and shift operations throughout the code for frequent divisions by two.

Ultimately, the intended use of this polynomial multiplication routine is within a 'powermod' function, which computes the result of an operation comprising some term, a, raised to a positive integer, n, modulo another arbitrary term, b. The terms a and b may be integers, polynomials with integer coefficients, or any combination of both. An algorithm for a typical *powermod* function resembles the one given below; in this instance, a is a univariate polynomial, $a(x)$, and b is a positive integer.

Algorithm 5.1.3 A *powermod* algorithm variant which computes $p(x) = a(x)^n \mod b$ in $O(\log n)$ time. This version has been adapted from the widely known integer *powermod* routine and is intended for use with the `ZZX` module in NTL.

```
1: procedure POWERMOD($a(x)$, $n$, $b$)  ▷ Input: A polynomial, $a(x)$, its exponent, $n$, and
   the integer modulus, $b$.
2:     $p(x) := 1$                                  ▷ Initialise the answer, $p(x)$.
3:     while $n > 0$ do
4:         if $n$ is ODD then
5:             $p(x) := p(x) \times a(x) \mod b$
6:         $a(x) := a(x) \times a(x)$
7:         $n := n/2$
8:     return $p(x)$;
```

The *powermod* algorithm is the final piece of the puzzle for a computational implementation of AKS in the form shown in Algorithm 2.4.2. By using it in combination with binary segmentation as the method for computing the two polynomial multiply operations therein, the $(x + a)^n \pmod{n, x^r - 1}$ term on the left hand side of Equation 2.4.1 can be resolved efficiently. *Powermod* operations involving a polynomial base for the exponent, as well as polynomial *and* integer moduli, is an intricate mathematical operation. To this end, extended versions of the algorithms shown in this section have been developed to accommodate this and are thoroughly documented in Section 5.2.1.

5.2 Congruence Testing

Recall from Section 2.4.2 that the reasons behind implementing binary segmentation are to leverage fast integer multiplication algorithms, which may have fewer overheads than those which are purely polynomial oriented, and explore whether it is a worthwhile alternative to the typical Karatsuba, FFT- or CRT-based methods.

In the congruence testing stage of Algorithm 2.4.2, the polynomials being worked on are of degree $r - 1$ and have up to r terms with coefficients of a size equal to n, the input being tested for primality. As n becomes very large, so too does r and number of operations involved in computing $(x - a)^n \pmod{x^r - 1, n}$.

This section covers the application of binary segmentation in practice, within the congruence testing stage of the AKS program, and the changes the original algorithm has undergone to implement it for this purpose. In addition, performance profiling of this untested binary segmentation-*powermod* pairing is conducted to assess the run-time, speculate on optimisation techniques, and benchmark it against comparable NTL routines. Finally, work conducted on implementing a rediscovered algorithmic approach to optimising the program concludes the section.

5.2.1 Applying Binary Segmentation

With working implementations for multiplying regular polynomials, using binary segmentation, and *powermod* operations, additional adaptations must be made if they are to work with polynomials modulo some integer h. Though their insertion into the congruence testing phase of the AKS algorithm, the Z variant of AKS becomes a reality.

Algorithm 5.2.1 A *powermod* algorithm variant which computes $s(x) = f(x)^n \mod g(x)$ in $O(\log n)$ time. This version is based on the widely known integer *powermod* routine and is intended for use with the `ZZ_pX` module in NTL.

```
1: procedure POWERMOD($f(x)$, $n$, $g(x)$) ▷ Input: A polynomial, $f(x)$, its exponent, $n$,
   and the polynomial modulus, $g(x)$.
2:     $s(x) := 1$                                   ▷ Initialise the answer, $s(x)$.
3:     while $n > 0$ do
4:         if $n$ is ODD then
5:             $s(x) := s(x) \times f(x) \mod g(x)$
6:         $f(x) := f(x) \times f(x) \mod g(x)$
7:         $n := n/2$
8:     return $s(x)$;
```

As necessitated by Equation 2.4.1, the algorithm given above can accommodate operations of the form $f(x)^n \pmod{h, g(x)}$ provided that the exponent n is a positive integer (as normal), the polynomial inputs have the form $f(x) \pmod{h}$ and $g(x) \pmod{h}$, and a compatible polynomial multiplication routine is used. In light of this, further modifications to the binary segmentation multiply algorithm are required and have been applied in Algorithm 5.2.2. In practice, the forms $f(x) \pmod{h}$ and $g(x) \pmod{h}$ are represented in C++ as `NTL::ZZ_pX` types, where the $\bmod h$ operation is implied, and passed as arguments to the function as such. Consequently, this is reflected in the notation of Algorithm 5.2.2 and the only noticeable changes may be found in lines 7 and 9 of the construction stage.

5.2.2 Performance Profile

After this new version of binary segmentation for polynomials modulo h (Algorithm 5.2.2) and the *powermod* routine for the same type of polynomials (Algorithm 5.2.1) were combined, After discovering that this combination was an order of magnitude slower in practice than the `NTL::PowerMod` function, tests were conducted on *powermod* implementation with binary segmentation to investigation and profile its performance.

Algorithm 5.2.2 Another variant of the binary segmentation algorithm, further modified to accommodate polynomials modulo some integer, h, as inputs. Note the $\bmod h$ operations in lines 7 & 9. Notation and symbolic representation is unchanged from Algorithm 5.1.1. Code implementing this algorithm in C++ may be found in Appendix C.

```
1: procedure MULTIPLY($f(x)$, $g(x)$)    ▷ Input: Two polynomials, $f(x)$ and $g(x)$, both $\bmod h$.
2:     Choose $b$ such that $2^b - 1 > \max\{U, V\} \cdot \max\{f_j\} \cdot \max\{g_k\}$    ▷ Initialise.
3:     $F = f(2^b - 1)$    ▷ Create binary segmentation integers.
4:     $G = g(2^b - 1)$
5:     $m = F \cdot G$    ▷ Integer multiplication.
6:     for each $i \in \{n \mid 0 \le n < U + V - 1\}$ do    ▷ Reassemble coefficients into signal.
7:         $s_i = \lfloor m/(2^b - 1)^i \rfloor \mod 2^b - 1, h$    ▷ Extract next $b$ bits.
8:         if $s_i > \frac{1}{2}(2^b - 1)$ then
9:             $s_i = s_i - ((2^b - 1) \mod h)$
10:            $m = m + (2^b - 1)$
11:    return $s(x) = \sum_{i=0}^{U+V-2} s_i x^i$    ▷ Base-$b$ digits of $m$ are desired coefficients.
```

To this end, it was hoped that areas ripe for optimisation would be identified and the runtime performance made more competitive with that of NTL. For performance profiling purposes, the binary segmentation program seen in Algorithm 5.2.2 can be broadly broken down into five sequential stages as follows:

1. **Initialisation** - encompasses all the operations involved in line 2, of which there are quite a few such as finding values for the $\max\{U, V\}$, $\max\{f_j\}$, and $\max\{g_k\}$ terms as well as the `while` loop used to evaluate the inequality and obtain b.

2. **Evaluation** - comprises the two substitution operations in lines 3 and 4, where $f(x)$ and $g(x)$ are evaluated for $2^b - 1$ from the previous stage.

3. **Multiplication** - only includes the big integer multiplication step in line 5.

4. **Construction** - involves the sequence of operations in the `for` loop in lines 6-10, where the coefficients of the polynomial product are constructed from the signal obtained from the integer multiply in the previous step.

5. **Return** - concerns the post processing operations in line 11 to return the polynomial product in the required form.

The $(x - a)^n \pmod{x^r - 1, n}$ term was replicated using a range of realistic values for a and r obtained from 25 values of n, ranging from 3 to 42 decimal digits in length. These parameters were passed to the `powermod` function in Algorithm 5.2.1, where each of the sections, described above, were individually timed over the course of the computation. This was repeated six times for each value of n and the times recorded as an average over these six runs, in what was a completely automated procedure through the use of a bespoke test program. The $\log_{10}$ of the times for each section has been plotted against several variables in Figure 5.1, Figure 5.2, and Figure 5.3 in an effort to understand the factors influencing the runtime of the program. It can be immediately seen that the proportion of time taken for the *initialisation* and *return* phases are insignificant relative to the others and so they are generally omitted from this discussion.

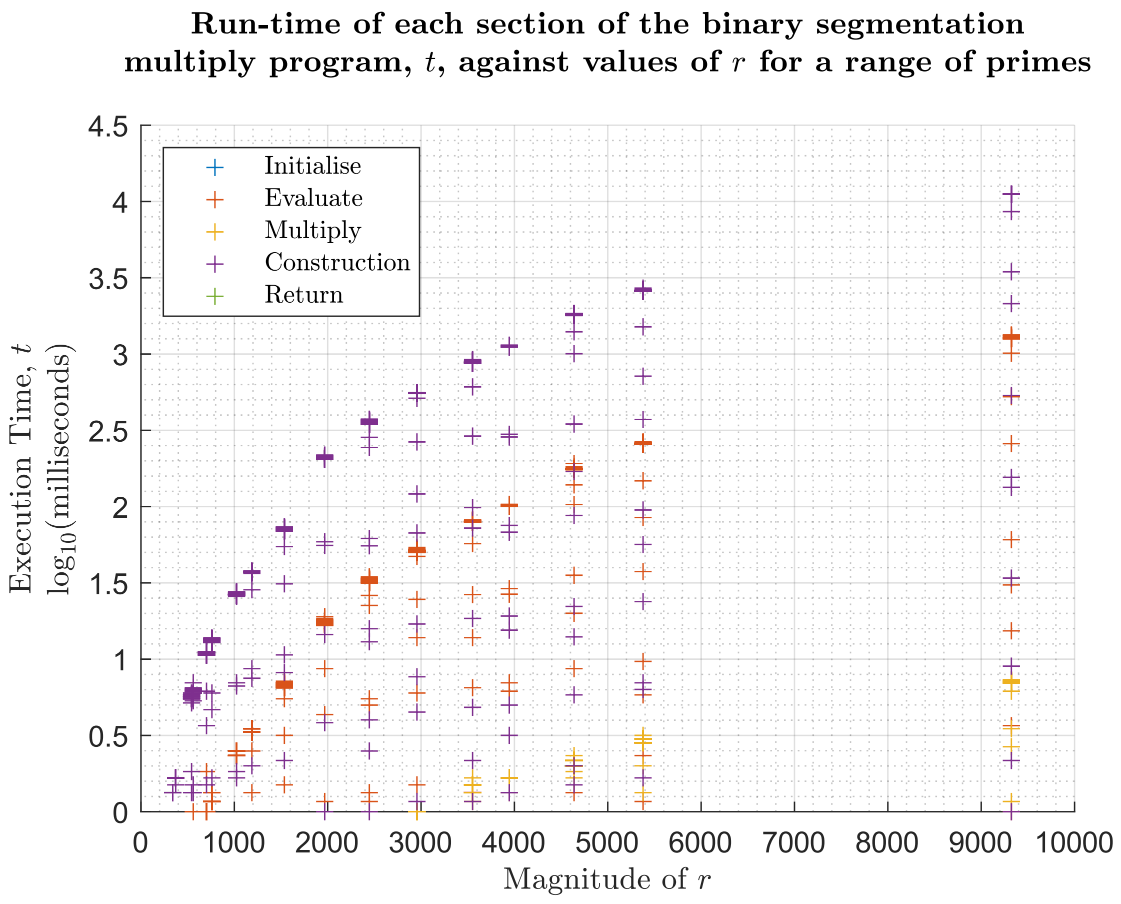


Figure 5.1: A scatter plot showing the effect of increasing r on the run-time of the different sections of the binary segmentation program, t. Note that colours appear deeper than depicted in the legend where points in that region appear in high concentrations.

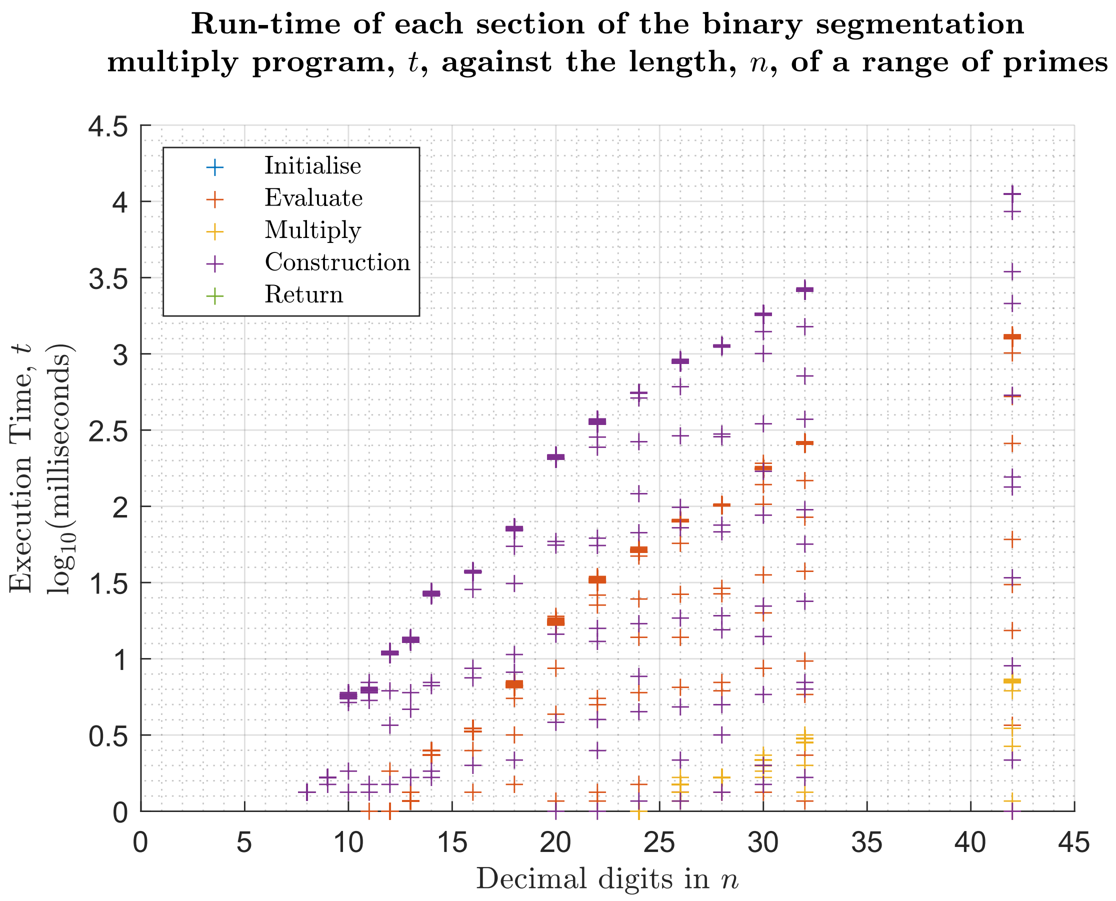


Figure 5.2: A scatter plot showing the relationship between the number of decimal digits, n, in the exponent of the polynomial term and the run-time, t, of each section of the binary segmentation program. Note that high concentrations of points appear in deeper colours than depicted in the legend.

The vertical spread seen in Figure 5.1 and Figure 5.2 is attributed to the early multiplication operations on polynomials of relatively small degree. The bold regions, where points are placed in high concentrations, should be interpreted as the trend line for the worst case run-times since these originate from operations on polynomials of a much larger degree. A high concentration of points is an artefact of the `powermod` function, used to profile the procedure, whereby any operation using Algorithm 5.2.1 eventually reduces to repeated squaring of a polynomial of the same degree. In practice this repeated squaring has a roughly constant run time and so its growth rate slows. Figure 5.3 depicts this more clearly, without the vertical point spread, by charting the time of the multiply operations based on the degree, D_s, of the output polynomial, $s(x)$.

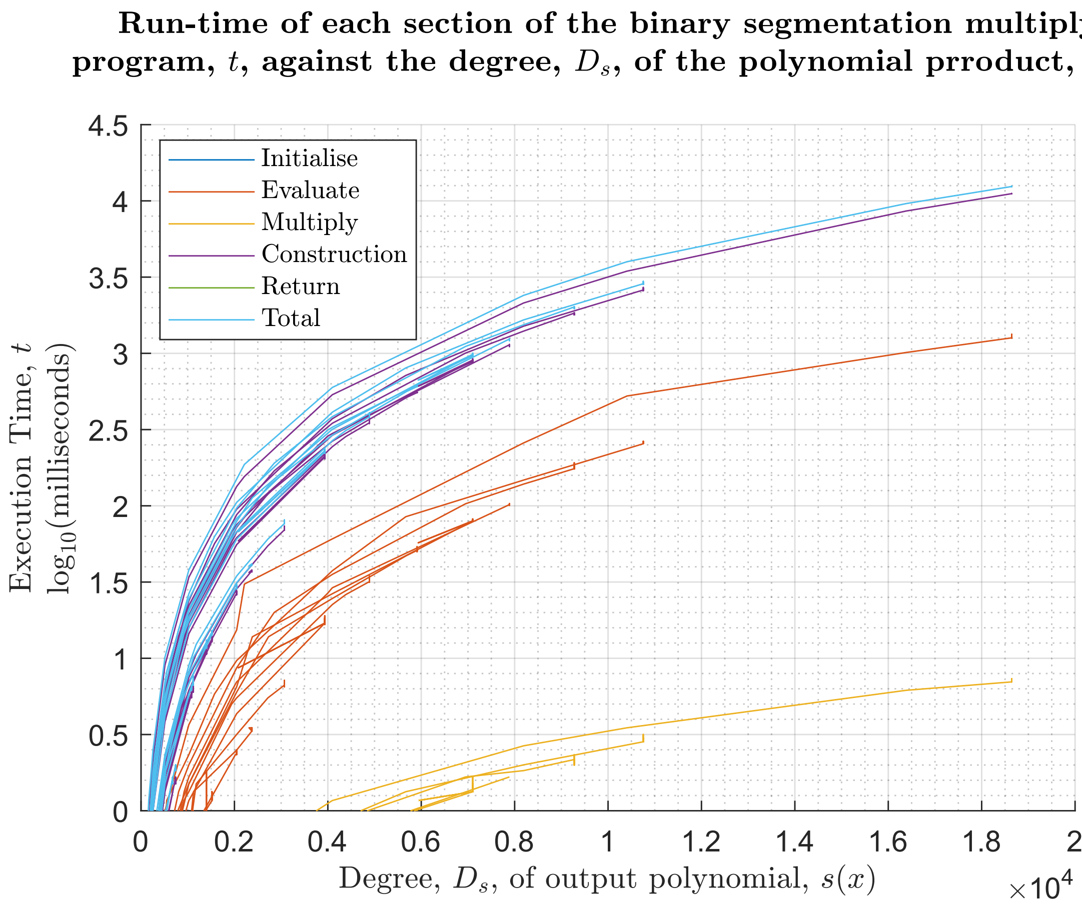


Figure 5.3: A line graph charting the run-times of different section of the binary segmentation program with increasing degree of the polynomial product, $s(x)$. This charts especially highlights how tightly the total time for the operation is bound to the polynomial construction time. Note how the integer multiply step grows, as well as how initialisation and return times are relatively insignificant.

In all the figures seen here, the two (arguably three) significant contributors to the runtime of binary segmentation are the *evaluation* and *construction* phase. The first of these, the *evaluation* phase, had already been correctly identified as a weakness in Section 5.1.2 and is to to with the bespoke helper function (see instance of `evaluate()` in Listing C.6) created specifically to compute the substitution since no such function existed in NTL. Optimisation of the function currently implemented has resulted in gains of up to 65% for moderately sized inputs. However, this is not to say that there is not scope for more improvements to be made since the *evaluation* stage still accounts for 8-11% of the total runtime for integers between 32 and 42 decimal digits long.

The runtime contribution from the *construction* phase is an order of magnitude greater still and is the more troubling of the two. Figure 5.3 shows how tightly coupled the *construction* time is to the total time taken for the binary segmentation procedure in its entirety. In fact, the proportion of time spent extracting the coefficients from the integer signal levels off at 90% of the total.

Furthermore, even resorting to parallel computing is difficult since the loop used to construct the polynomial at this stage of the procedure is a very serial process, where each iteration is dependent on the value of m from the preceding one to correctly handle overflow between the coefficient cells. This lack of data-level parallelism means the only hardware acceleration option is to use a faster or more powerful CPU.

The performance of regular binary segmentation in practice would indicate that computational arithmetic over the integers is generally worse than over polynomials, with the overheads of going to and from polynomials and integers respectively a factor of 14 rather than the factor 2-4 initially expected. Nevertheless, areas for optimising binary segmentation have been experimented with and consist of assumptions about the polynomials themselves as well as a re-examination of the *powermod* function.

To completely avoid some overheads altogether and recalling that the size of the coefficients being dealt with are about the size of n, an approximation for the product $\max\{f_j\} \cdot \max\{g_k\} \approx n^2$ can be made for input polynomials of very large degree. The variant of the binary segmentation routine implementing this is referred to as the approximating (*Approx.*) version and its effect on performance, or lack thereof, is seen in Figure 5.4, Figure 5.5, and Figure 5.6, and discussed in the next section.

Earlier, it was said that the *powermod* function reduces to a squaring operation once the exponent itself reduced to a power of two. To use an example, computing $(x - a)^n \pmod{x^r - 1, n}$ using Algorithm 5.2.1 for a 32 digit value of n consists of 18 multiplications followed by 134 squaring operations. With this in mind, there is scope for stripping out many of the steps needed to compute the product of two *different* polynomials to produce yet another version of binary segmentation which is tailored specifically toward squaring operations.

Whilst the development of a tailored version such as this was not completed, the potential impact of using an efficient polynomial squaring algorithm was simulated using the `NTL::sqr` function for the squaring step in line 6 of Algorithm 5.2.1. A variant of binary segmentation like the one described here is referred to as the squaring (*Sqr.*) version and its effect on performance is also seen in Figure 5.4, Figure 5.5, and Figure 5.6, and discussed in the next section.

As anticipated from the theoretical time complexity conjectured in Section 5.1.2, the best indicator for the runtime of the binary segmentation multiply routine in practice is the degree of output polynomial product. Furthermore, this property can even be predicted using simple mathematical facts about polynomials without having to compute the product at all; if $f(x)$ has degree D_f and $g(x)$ has degree D_g, then the degree of their product, D_{fg} is equal the sum of D_f and $g(x)$. Should binary segmentation ever be used in a polyalgorithm with other polynomial multiply routines, this property of polynomials could be used a heuristic for the performance one can expect from it when deciding which algorithm to use from the set.

5.2.3 Binary Segmentation versus NTL

NTL uses four different algorithms, within a polyalgorithm, for multiplication of univariate polynomials with integer coefficients. The choice of algorithm is heuristic and based on the degree and size of the coefficients. The classical algorithm and Karatsuba are used for smaller polynomials while Schönhage and Strassen (1971) is used for polynomials with 'huge coefficients and moderate degree', and CRT- or FFT-based methods are used for ones with 'moderate coefficients and huge degree' (Shoup, 2001b). Given the size of the coefficient and the degree of the polynomials encountered in this application, it is safe to say that one or both of the latter two is being used at any one time when using NTL for polynomial multiplication.

At the same time that the binary segmentation procedure was being timed within its *powermod* function pairing, the same tests described in the last section were conducted using the `NTL::PowerMod` function and timed. These times are used as a benchmark, against which the performance effects of minor modifications to the regular binary segmentation-*powermod*, for approximating and squaring, could be evaluated. The performance of the different computational *powermod* functions are compared in Figure 5.4, Figure 5.5, and Figure 5.6.

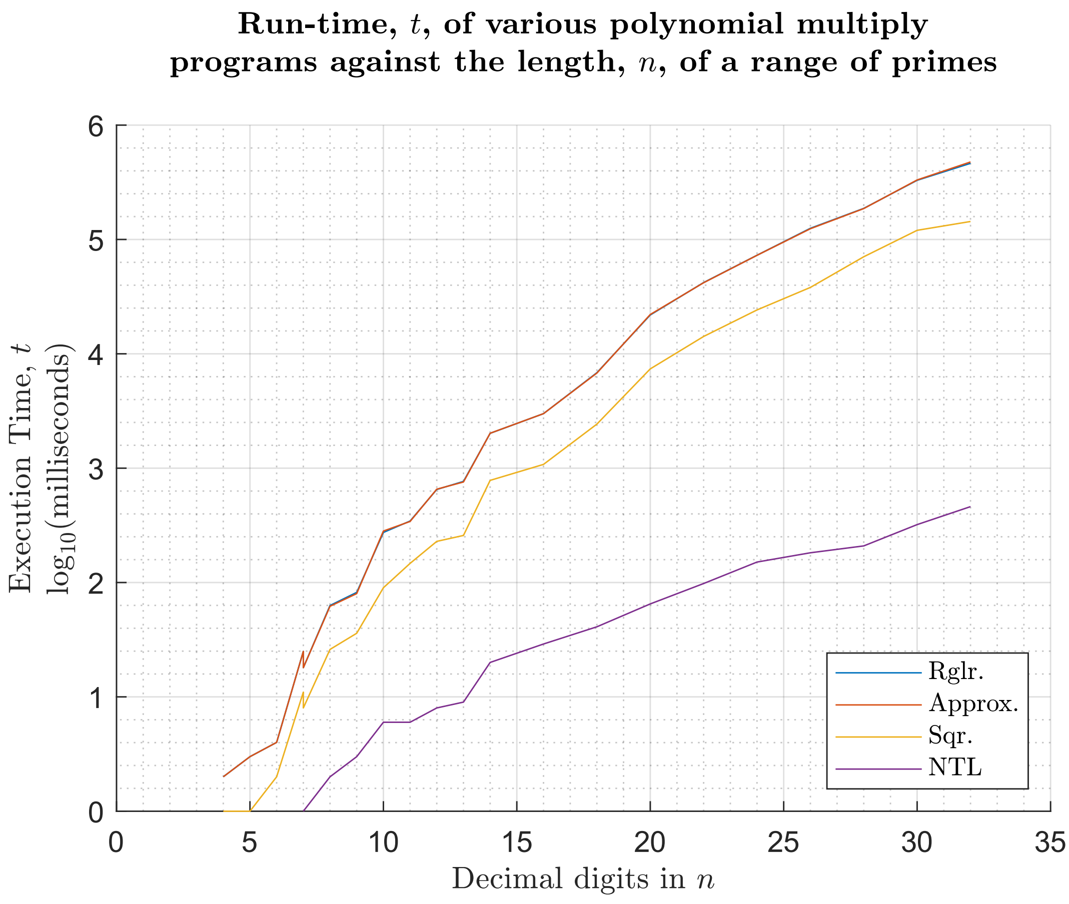


Figure 5.4: Comparing the runtime performance of the heuristic polyalgorithm used by NTL for polynomial multiplication with those of the numerous binary segmentation routines. The number of decimal digits, n, in the exponent of the polynomial term is used as the independent variable. Note how the lines for the regular and approximating variants are overlapping, for they are apparently indistinguishable in performance for integers of this size.

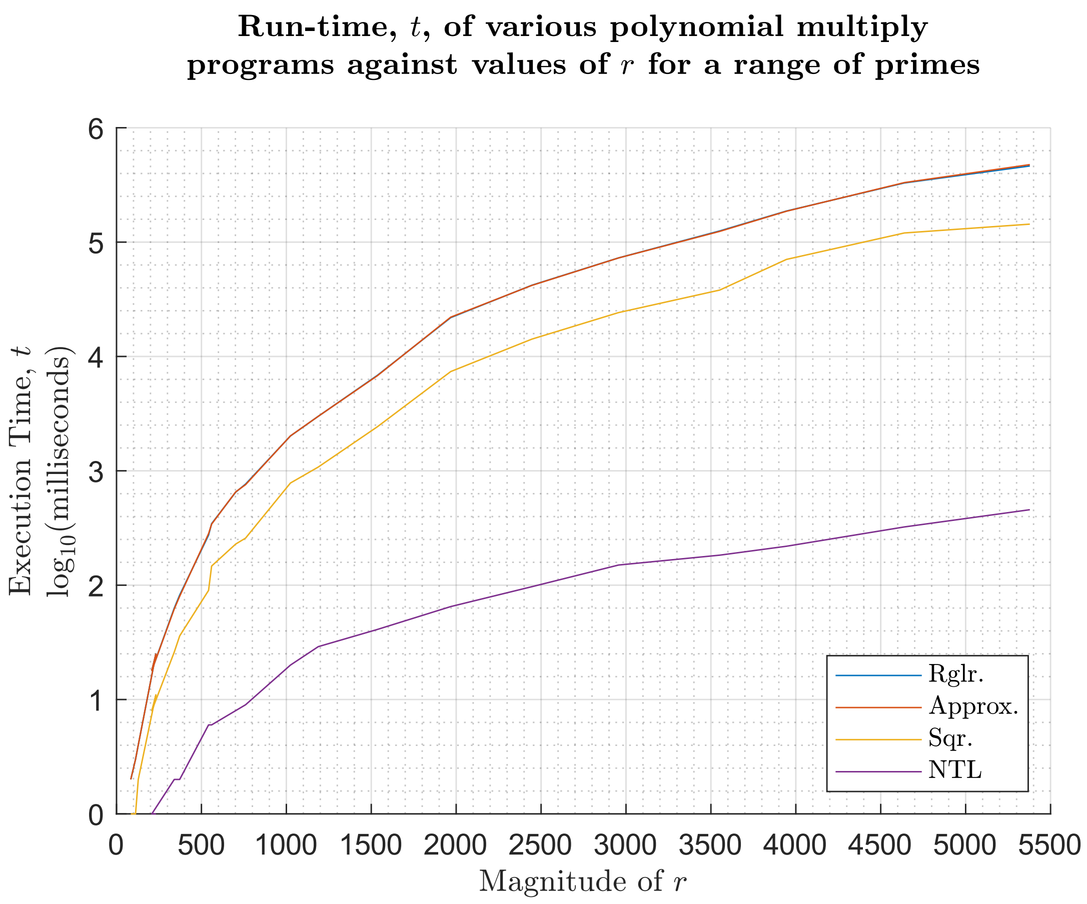


Figure 5.5: Comparing the runtime performance of the heuristic polyalgorithm used by NTL for polynomial multiplication with those of the numerous binary segmentation routines. The values of r, obtained from the exponent of the polynomial term, are used as the independent variable. Note how the lines for the regular and approximating variants are overlapping, for they are apparently indistinguishable in performance for integers of this size.

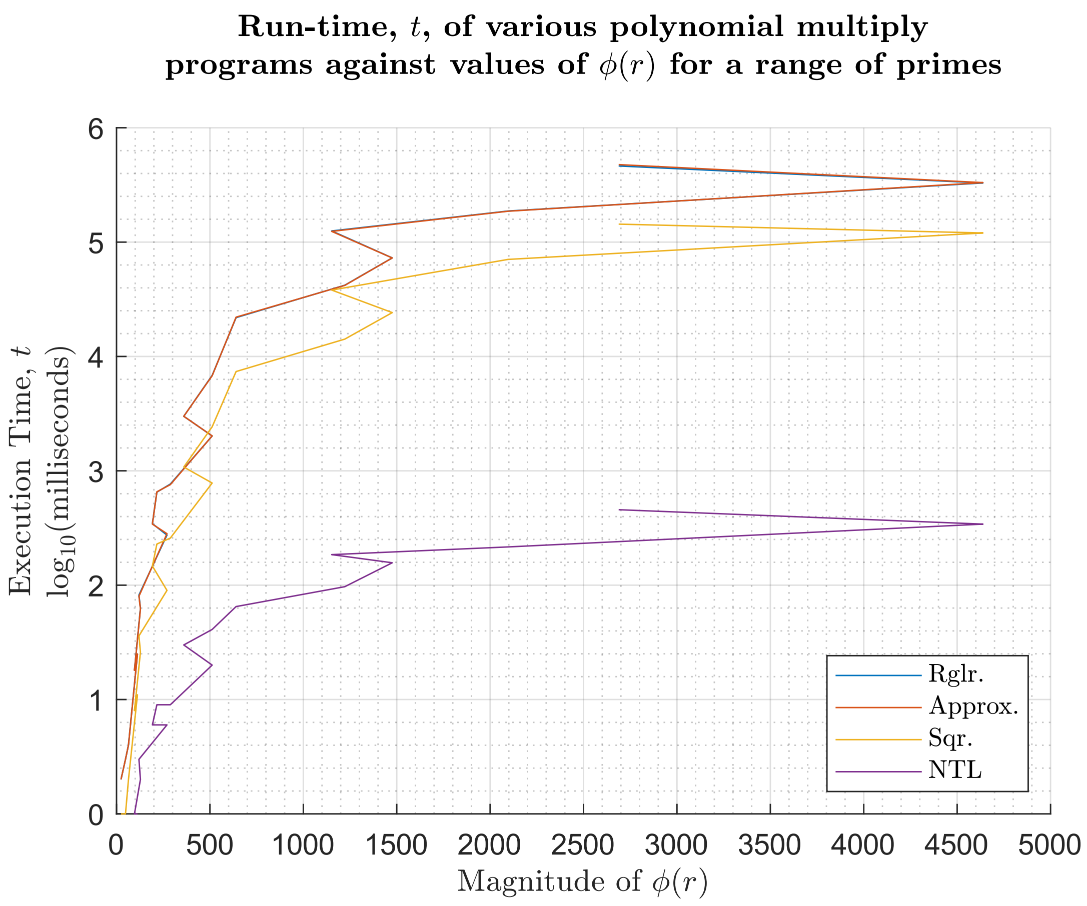


Figure 5.6: Comparing the runtime performance of the heuristic polyalgorithm used by NTL for polynomial multiplication with those of the numerous binary segmentation routines. The values of $\phi(r)$, obtained from the exponent of the polynomial term, are used as the independent variable. Note how the lines for the regular and approximating variants are overlapping, for they are apparently indistinguishable in performance for integers of this size.

To put it in no uncertain terms, the `NTL::PowerMod` function utterly outperforms any binary segmentation-based *powermod* variant, with its runtime performance clocking in several orders of magnitude faster in every test when assessed over a range of dependent variables. Nevertheless, some useful insights have been uncovered from this assessment.

Approximating a would not appear to be beneficial in any way, and even detrimental in some cases, for numbers of this size. Although the effect of this modification is measurable in the low hundreds of milliseconds, it is generally insignificant relative to the total runtime of the procedure, much like the rest of *initialisation* phase as a whole, within which this change resides. Furthermore, Figure 5.5 would indicate that the runtime, t, is more closely related to r than $\phi(r)$ or n, which display far more variability in their profiles. This implies that it may be possible to express t as a function of r, but the precise nature of this relationship is not fully understood and the question of specifying precisely if remains open.

The potential time savings from the use of a specialised squaring function can be delivered in reality. The performance improvements from the `NTL::sqr` function are significant considering the $\log_{10}$ scale used for the execution time in the figures above, although it must be said they are sightly less than expected perhaps. Despite this, the use of a specialised squaring function for polynomials of the form $f(x) \pmod{h}$ and could, and perhaps should, be explored further if binary segmentation is to become a viable and competitive method.

Finally, the NTL source code might lend some insights into the methods used in its own *powermod* implementation which could then be applied to practical binary segmentation programs. It is also, worth remembering that the NTL and GMP libraries are prolific and something of a gold standard in the realm of software libraries for serial integer arithmetic on CPUs. Attempts should be made to bring the performance of Binary segmentation should be brought in line with NTL before any further potential enhancements are made using other libraries such as FLINT (Harvey and Hart, 2007) or MIRACL (Scott, n.d.).

While binary segmentation may be unlikely to ever come close to the performance of NTL (there must be a reason it is hardly featured anywhere), a useful exercise might be to assess its performance more thoroughly. Doing so to greater extent than that seen in this work would establish with more certainty whether it has any niche applications, or where it could sit in the arsenal of a polyalgorithm for multiplying univariate polynomials with integer coefficients, if at all.

5.2.4 The Carmichael Lambda Function

After seeing the performance of the binary segmentation in practice, a renewed approach to the algorithm itself was taken in an effort to seek more significant performance gains, rather than pursue incremental optimisations any further. Egan (2005) hypothesised that the final loop in the AKS variant of Lenstra (2002) only requires $\lambda(r) - 1$ iterations, rather than the original number of $\phi(r) - 1$. She showed that this small, but important, change to algorithm reduced the runtime by nearly a factor of five, on average, and up to a factor of 36 at best.

To give Egan's results some context and treat them with the appropriate caution, it must remembered that this five-fold speed up on average includes the possibility of no speed up at all. Section A.2 demonstrates how volatile the relationship between $\lambda(n)$ and $\phi(n)$ can be generally. To put this phenomenon in perspective from a practical standpoint, consider the 42 decimal digit (139-bit, or three 64-bit word) prime, which took 3 hours and 45 minutes to produce a serial verification, and for which $r = 9323$; since $\lambda(r) = \phi(r)$, the same computational work is required so there could be no algorithmic performance gains.

Cases such as this grounds the potential speedup factor of 36 in reality, but despite the high variability, the best runtime performance gains are still significant and would pay off in cases where $\lambda(r) < \phi(r)$, and therefore not having to compute Equation 2.4.1 as far as $a = \phi(r) - 1$, when verifying very large primes. With this in mind, recalling its definition (Definition 2.1.7), formula (Equation 2.1.4), and conditions (Equation 2.1.5), an algorithm for the Carmichael lambda function can be given below and the C++ program implementing it may be found in Listing C.7 of Appendix C.

Algorithm 5.2.3 Algorithmic representation of Carmichael's Lambda Function, or reduced totient. Code implementing this algorithm in C++ may be found in Appendix C.

```
1: procedure LAMBDA($n$)                                    ▷ Input: $n \in \mathbb{N}, n > 0$.
2:     if $n$ is PRIME then
3:         return $\phi(n)$;
4:     Decompose $n$ into a sequence of unique prime factors, $p_i$, and exponents, $\alpha_i$:
           $$n \equiv \prod_i (p_i^{\alpha_i})_{i,k\in\mathbb{Z}^+}^k, \text{ where } (p_i^{\alpha_i})_{i=1}^k = (p_1^{\alpha_1}, \ldots, p_k^{\alpha_k})$$
5:     for $j \in \{l \mid 1 \leq l \leq |(p_i^{\alpha_i})_{i=1}^k|\}$ do
6:         if $\alpha \leq 2$ or $p \geq 3$ then
7:             $(\lambda(p_{i=j}^{\alpha_{i=j}}))_j := \phi(n)$
8:         if $\alpha \geq 3$ and $p = 2$ then
9:             $(\lambda(p_{i=j}^{\alpha_{i=j}}))_j := \frac{1}{2}\phi(n)$
10:    return $LCM((\lambda(p_j^{\alpha_j}))_{j=1}^k) = LCM(\lambda(p_1^{\alpha_1})_1, \ldots, \lambda(p_k^{\alpha_k})_k)$;
```

5.2.5 Performance Optimisation in C++

The C++ programming language is regarded as very efficient in the way that it makes use of hardware resources and preferred over higher-level languages for application where high-performance is an important requirement. However, sloppy C++ code may be no more efficient than an equivalent program written in a higher-level language so, to take advantage of its strengths and ensure it is well optimised for runtime performance, strong knowledge of the language is required.

Ignoring performance gains at this level would only serve to mitigate any benefits achieved by deploying it in parallel with GPGPU. As such, whilst at times it may be implied or taken for granted, the author affirms that the aggregation of these 'micro optimisations' can yield significant returns, whose worth cannot be understated. Already having a competent understanding of C programming, he has become well acquainted with C++ over the course of completing this project and the language-specific techniques employed to optimise the relevant parts of the codebase are described explicitly as follows.

Function calls incur overheads which can accumulate if a function is called inside a loop, as demonstrated below in Listing 5.1. Such overheads can be avoided by simply moving the loop within the function itself, as in Listing 5.2, thus limiting any overheads to those incurred by a single function call. For small iterators and small functions, the performance gains achieved in this way will be insignificant but could be substantial as the iterator become large. The function `CongruenceZnx`, seen here specifically, is the embarrassingly parallel section of the program. Conveniently, writing it in the form seen in Listing 5.2 facilitates its parallelisation in CUDA by encapsulating it in a single function and later launched as a kernel on the GPU.

Listing 5.1: Function is called in every iteration of the loop. Also note the prefix form of the increment in the loop declaration and the arguments passed by constant reference.

```
long CongruenceZnx(const ZZ& n, const ZZ& r, const ZZ& r2){
    ...
}

long a = ...;
for (long j = 1; j <= a; ++j) {
    long f = CongruenceZnx(n,r,r2);
}
```

Listing 5.2: The loop is moved into the function body and executes with a single function call. Also note the prefix form of the increment in the loop declaration and the arguments passed by constant reference.

```
long CongruenceZnx(const ZZ& n, const ZZ& r, const ZZ& r2, const
   long& a){
    for (long j = 1; j <= a; ++j) {
        ...
    }
}

long a = ...;
long f = CongruenceZnx(n,r,r2,a);
```

Furthermore, one can also reduce the overheads in the loop itself. Note that the iterators in the loop declarations in both Listing 5.1 and Listing 5.2 above have the prefix form `++variable`, instead of the traditionally taught postfix form `variable++`. Just as with the function calls, this might be insignificant for small iterators, but, as `a` tends to become very large, which it may for large inputs of n to Algorithm 2.4.2, the difference, however small, is quantifiable nonetheless.

Another way of reducing the function call overhead is to consider *inline functions*. The inline feature is useful for very small functions where the calling time is significant relative to the execution time of the function. Essentially, the runtime performance gain is achieved by substituting the code from the inline function body into the point in the program at which it is called. A function can be made 'inline' using the `inline` keyword at the start of the function declaration before the return type.

The compiler, however, will not interpret this as a command, but as a request and may ignore it if the function is recursive, contains a loop, static variables, or a non-void return type. As such, whilst inline functions can be used relatively liberally, not all will see any benefit from receiving this treatment and, according to Ruderman (2021), may even suffer in performance if misused.

The final method of reducing function overheads is to pass the parameter arguments by constant reference, rather than by copy. When calling a function using pass-by-copy arguments, e.g. `long a`, a copy is made of the variable which adds complexity in both time and space unnecessarily. For large parameters, the additional complexity costs incurred may accumulate and grow substantially. This can be avoided using pass-by-reference, which uses reference arguments to refer to the memory address of the parameter (typically only a few bytes), rather than copying the parameter itself. To pass objects by reference, the arguments in the function declaration must have the `&` operator in postfix with the type declaration of the variable, such as `long& a`.

However, using reference parameters can cause further problems related to mutability and code readability. Passing values by reference in the fashion described above implies that the function may be modified by the function. The reference can be safely made immutable by making it constant with the `const` keyword at the beginning of the parameter declaration which, usefully, will produce a compile-time error if there is any attempt to re-assign it in the function later on. For constant references, the arguments in the function declaration will appear as `const long& a`, for example.

Lastly, as a general rule for command line output, the `printf` function inherited from the C programming is favourable over the common C++ equivalent, `std::cout`, which uses the `iostream`. A trivial program to test and compare the two will reveal that `printf` execution time is as much as 12 time faster, which is significant. In addition to being potentially faster and familiar from C, `printf` commands are generally shorter and makes for more readable code meaning it is a highly appropriate alternative.

Since the amount of data being output is relatively low, the reader will observe that both `printf` and `cout` commands are used throughout the codebase for this work. The author avoids performance penalties by using `printf` for print statements when performance is key but console output is still desired, and `cout` commands, which are strictly kept outside time-critical sections of code, for file outputs. Given that I/O can be a bottleneck anyway, this strategy is an acceptable compromise.

Chapter 6

Parallel Computing

The NTL-*CUDA* compatibility oversight was unfortunate, but should not be treated as a permanent impasse for future work. Furthermore, work could still be done on validating the strategies proposed and presenting a proof-of-concept, albeit on a far smaller scale than originally intended, using the NTL basic thread pool feature to implement and test AKS programs in parallel. This chapter gives an account of implementing AKS in parallel on multi-core CPUs and lays out recommendations for any future implementations of AKS in *CUDA*.

Use of the word 'basic' as a descriptor for multi-threading functionality within a software library is somewhat self-deprecating. Whilst it might be basic relative to *CUDA* C/C++, this should not be seen as a disadvantage. The way in which the NTL thread pool interface abstracts much of the technical aspects involved (such as memory allocation), whilst simultaneously delivering a learning experience in parallel computing, should be praised.

6.1 Parallel Computing Strategies for AKS

Much has been said about parallel strategies for exploiting data level parallelism at a general level. This section details a strategy specifically intended to target the hardware and the sheer volume of computing resources (GPU cores) at the disposal of the main program. The strategy proposed is similar to many of the common pre-existing *CUDA* applications centred around array operations or vector arithmetic. Suppose the values in the domain of a are listed sequentially in an array, and passed to the device such that each element of the array is assigned to a separate thread tasked with computing a congruence test.

The output from each thread would be an element of a new, second array, equal to the length of the original array, where each element is a one or a zero corresponding to the outcome of the congruence test on the corresponding value of a from the original array. At the end of the parallel computation, the second array is copied back to the host from the device and checked: if the output array contains any zeros (or the sum of the array is less than/not equal to its size) then the input n is NOT PRIME. Otherwise, if the array only contains ones, n is PRIME.

This would be relatively straightforward to implement, but not necessarily the fastest; one must wait for all the GPU threads to terminate 'naturally' before proceeding with additional serial computations for searching/summing the array to produce the solution, which would, in turn, incur runtime penalties. Only one congruence test need fail to determine compositeness (or NOT PRIME), so a faster strategy would terminate all other concurrent threads as soon as any one of them produces a 'negative' result, or do away with the array method altogether.

GPU architecture mandates that all threads have to be doing the same thing at the same time, or idling. Knowing when to terminate the concurrent threads prematurely would have to involve a global signal of some kind which must be set in the event of a failed test. After that, all threads would need some way of periodically checking to see if they should terminate. Given that GPU global memory is relatively slow, repeatedly checking if any other thread have been flagged might incur overheads which may mitigate any time gains. These overheads would add to those already involved with launching the kernels to parallelise the program, including thread creation, memory allocation and transfer.

Given the novelty of applying GPGPU to AKS and the intricacies of producing a *CUDA* program which could behave in this way, early thread termination requirements were deemed to be non-essential and 'nice to have'. Pursuit of this purely runtime-centred functionality would be left for late-stage development or future work, and the chosen parallel strategy, as described here, would be taken further.

6.2 AKS in Parallel

6.2.1 Thread Pool Trials

Running the $Z_n[x]$ program of the Lenstra variant in parallel was achieved using the thread boosting features built-in to the NTL library and utilises multiple machine cores to accelerate the lower level computations. This feature supposedly makes factorising polynomials with big integer coefficients modulo some number, `ZZ_p`, up to 6 times faster when using eight cores. Whilst in reality the *NVIDIA CUDA* development toolkit would be the primary parallel computing API for deploying a program for GPGPU, it was hoped given the incompatibility, the NTL thread pools would still offer some valuable lessons which could be carried forward in future.

For this application, thread boosting features were implemented in accordance with precisely the strategy described in Section 3.1.1, whereby the domain of a would be evenly partitioned into c sub-domains depending on the number of available machine cores, c. In addition, elements of the strategy described in Section 6.1 were incorporated to simulate the strategy for GPUs and adapt it to suit the handful of cores on the CPUs available on a range of personal machines (two consumer desktops and a laptop, each with a very different set of hardware specifications which may be viewed in Section A.1). Although no GPUs were in use here, a very similar strategy would likely be implemented later in *CUDA*. As such, it would be replicated like-for-like as far as possible, but in miniature per se.

To implement the congruence testing in parallel. The final version of this parallel implementation is included for reference under Section C.5. Design changes made to the strategy for this implementation included substituting the array container originally proposed with a vector to simplify operations using the dedicated vector utility functions, such as `.size()`, `.push_back()`, and `.pop_back()`.

Using a limited selection of integers of no more than with 42 ten decimal digits to limit run time, improvements made were significant relative to a serial run time. Since the congruence test typically failed when $a = 1$ for the majority of composite integers tested, the program would often return a solution no faster than if it were running on a single core. However, the multi-threading capabilities came into their own upon the input of a prime and the congruence tests were computed must faster, and in a time roughly proportional to the number of cores on the machine, as expected.

As anticipated a phenomenon whereby each thread would continue to process the set of values for a it had been assigned, even if another congruent thread had finished the computation, halting either upon completion of the tasks assigned or after asserting that the congruence test was false. As a consequence, on an eight core machine, eight values of a would be returned for a composite input and the program would only terminate once the last of these had been found.

The lesson from this is that, to obtain optimal performance, there should be a requirement whereby the moment one thread has discovered a false congruence relation (and thus a solution to whether the input is PRIME or NOT PRIME) all remaining concurrent threads must be halted, terminated, or destroyed, in some way such that they do not incur any further runtime penalties.

Basic as they were, experimenting with NTL thread pools had introduced parallel processing in an accessible, uncomplicated manner. Perhaps most importantly, for such a brief amount of time spent using them, this experience was both efficient and worthwhile in terms of early exposure to parallel processing. Cautious of using unnecessarily large primes and sinking additional resources for testing a method of implementing the program in parallel that were never going to be retained, these endeavours were terminated once the author was satisfied that the learning opportunities with NTL had been exhausted, and confident in further pursuing high performance computing, this time using *CUDA*.

6.2.2 Strategy Validation

It remains to be seen whether this strategy scales sufficiently to be used when it comes to implementing it in parallel in a GPU. Tests were successfully conducted using this strategy on CPUs with up to eight cores and using moderately sizes primes up to 42 digits in length. The design of this strategy, and its implementation, accommodates scaling from four to eight cores without any difficulty.

Early, but incomplete analysis and profiling of the results may be seen in Section A.3. Notable results even at this early stage are the improvements in run-time performance due to the additional cores, as expected, but also the amplification of these gains through by using the Carmichael function to compute a.

6.3 Parallelism & GPGPU with CUDA C/C++

Recommendations for future work on applying GPGPU to AKS will now be made to close the topic of *CUDA* and GPGPU generally for this project. These are lessons from first hand experience of someone exploring the subject for the first time and generally centre around time efficiency and ease of deployment on a GPU, or GPU cloud such as the *Hex*. It goes without saying that conducting the necessary testing is a process which can be entirely automated, in most cases, using `bash` scripting and the shell. or command line, of the container which will time once initially set up.

The first is to consider having a *CUDA*-enabled GPU available locally (or via a remote desktop), either through a personal machine or, ideally, one provided by the institution where `sudo` or `root` privileges are also granted. The reason for this falls down to the difficult installation of the *CUDA* API itself, particularly on a machine sans *NVIDIA* hardware.

Ideally, one would want all the necessary compilers, debugging tools, and target device to develop and test a program for GPGPU. This is far easier to achieve if one has a machine with the *CUDA* API pre-installed and/or a *CUDA*-enabled device readily available. The device itself need not be top of the range, merely adequate for small validation tests and debugging prior to up-scaling the kernel block size for the intended devices (i.e. the GPUs on the *Hex* cloud).

Secondly, a *Docker* image for running a container on the *Hex* GPU cloud ought to pre-include *CUDA* at the very least. This will, again, save time by avoiding having to repeat the *CUDA* installation process. If there exists a *CUDA*-compatible software library at this point, then this should also be included on the image file. *Docker* images with pre-installed components are freely available.

Finally, it is strongly advised to generalise the kernel code to be ubiquitous or polymorphic to all *CUDA* devices. This can be done from a local, or lower compute capability, GPU so that once it is ready to deploy to a larger one, say on the GPU cloud, the program may be compiled for the device and no time is lost to having to refactor it to make it hardware specific. This refers mainly to the choices of grid and block sizes of the kernel, which is dependent on the number of thread required, which is, in turn, related to the magnitude of a from the main AKS program and will most likely be computed on the host.

Chapter 7

Discussion

7.1 Evaluation of Methods

The limitation for this work stem from the hardware resources used. Issues relating to the use of the inbuilt clock from the C++ standard `chrono` library and CPU throttling due to were not encountered to a large degree and did not prove to be an obstacle to meeting the research objectives under the proposed methods. The hardware specifications were limiting to an extent and future work would do well to use proper facilities such as `linux.bath` via SSH for prolonged tests in serial and parallel.

Under the circumstances, results produced from these methods are sufficient at this stage given that a higher degree of precision is not required here and conclusions may be drawn without ambiguity. However, this should not rule out the prospect of additional testing in future, either to validate these findings or benchmark newer implementations of the Z version of Lenstra's algorithm.

Planning of this research focused greatly on development, which has resulted in numerous successful implementations of mathematical functions, which have been extensively tested to ensure they are mathematically robust. Whilst these efforts were not misplaced by any means, some time should have been spared for a similar if not equal devotion to benchmarking and experimental analysis.

7.2 Evaluation of Work

Recalling the objectives set out in Section 4.1, the following judgements may be made against them. in the case of the first objective, all conditions have been satisfied in full - the Z version of the algorithm uses binary segmentation as the primary technique for polynomial multiplication and the performance of this variant relative to the $Z_n[x]$ version has been assessed. The second objective has been met to a lesser extent with limited exposure to parallel computing on GPUs. This is mitigated by robust implementation in serial and parallel in CPUs as well as a good understanding of CUDA and what must happen for this objective to be met. For the third objective, no judgement can be made in regards to the theoretical practicality of certifying large primes using GPUs, however plan for this project has been unsuccessful and could therefore be judged as impractical *for now*.

When evaluated against the intended outcomes set out in Section 4.2, the criteria of the first four, of eight, are met. Following the setbacks experienced with GPU compatible libraries, this work still meets the second alternative endpoints set out in Section 4.2.2 by leaving both implementations correct, profiles for performance, and in a state where that they may be ported across to *CUDA* for deployment on a GPU relatively without significant refactoring.

7.3 Conclusions

To conclude this work, the main contributions of this research are summarised. Highly optimised and CPU-parallel implementations of the $Z_n[x]$ version have been produced, and this version has been shown to the the more high-performing of the versions compared. The versions it was compared to are the numerous iterations of the novel Z version of the algorithm which includes the, now tested, binary segmentation algorithms for multiplication of univariate polynomials with integer coefficients.

Whilst the performance of this new Z version left something to be desired, the practical performance of the binary segmentation multiply technique underpinning the theoretical case for this version has been thoroughly profiled. Improvements for both binary segmentation, as well as the *powermod* routine within which it resides, have been suggested and left open to further work in future. Another contribution to this project is the renewed inclusion of Carmichael's reduced totient and the associated high performing source code. This value of this function to this application has been highlighted once again and can offer inconsistently significant gains in runtime performance when certifying very large primes.

In addition, a reliable parallel computing strategy specific to AKS has been proposed and implemented on a CPU for validation. Full scale general purpose implementations will almost certainly need to adapt and tailor it to the hardware being used but, at the very least, it offers a solid starting point from which to build on.

Finally, a niche demand, but a demand nonetheless, for an arbitrary integer arithmetic library, similar to NTL but for use on GPU device kernels, has been highlighted. It is firmly believed that GPGPU has the potential to open AKS to the real world by putting it into practice effectively, thereby bringing it into the modern era of computing. The demand will likely be met in time and once it has been, it is hoped this work will help deliver AKS implementations on GPUs.

7.4 Further Work

There remain a number of open questions and opportunities for further work related to this project. Many of these have been raised in the relevant places throughout this dissertation but the main ones are summarised in this section. Of course, the largest open question is to do with how AKS fares when implemented on a GPU, the answer to which has been hampered in this work by incompatible computational arithmetic libraries. In theory, AKS could still be implemented without library-enabled arbitrary precision integers and tested using smaller primes than originally intended. The recommendations for doing so based on some early experiences are laid out in

Section 6.3. This may not, however, fulfill the main purpose of this project as there are not likely to be many practical applications for AKS for certifying primality of primes limited in size, but, it may be a way of exploring the scalability of the parallelism to GPGPU and multiple GPUs.

Much of this research has explored whether binary segmentation is a viable and credible method for computational multiplication of univariate polynomials with integer coefficients. At present and based on the evidence seen here, the answer would be 'no', but this is not to say that can never change. Several proposals for accelerating it, to the point where it is viable and competitive with other methods have, been put forward and comprise suggested contributions to the NTL library as well as marginal optimisations to accrue more incremental gains.

The variant from Lenstra (2002) was chosen for its ease of implementation in practice rather that for its speed, and there are better options in terms of time complexity. The Z version is redundant until the performance of binary segmentation improves significantly. As such, any further work in the near-future should focus on the $Z_n[x]$ version, other variants of AKS, or the other algorithmic optimisations, as well as binary segmentation, suggested by Crandall and Papadopoulos (2003) which have not been implemented here. For serial computing, other libraries, such as FLINT (Harvey and Hart, 2007) and MIRACL (Scott, n.d.) should be recognised as equals of the established NTL and GMP libraries. FLINT and NTL are well matched and their performance is compared and documented in the respective documentation files. The work of Gallot (2005) is testament to the capabilities of MIRACL in his implementations of AKS.

From the literature studied, it is perceived that work in the field of primality testing has stagnated in recent years as it did in the late 20th century. It stands to reason that this may be perhaps due, in part, to a lack of demand for further research; a longstanding open problem has been closed, for some time now, and pre-existing practical algorithms for primality proving are fast and robust enough. This lends the question: what does the future hold for the field of primality testing?

Does its future lie in polyalgorithms and hybrid implementations, or in brute-force parallel computing? Will the advent of quantum computing change the way theoretical computer scientists define what it means for an algorithm to be computationally feasible? If so, what happens to the existing systems built on untractable problems, such as the factoring problem or the discrete logarithm problem, once they become tractable? Such questions are something of a rabbit hole, but the reader gets the point. Consequently, it is hoped that this work has delivered meaningful contributions, generated results for debate, answered some questions and perhaps produced more of them.

Bibliography

Aaronson, S., 2003. *The Prime Facts: From Euclid to AKS* [Online]. Available from: `https://scottaaronson.com/writings/prime.pdf`.

Adleman, L.M., Pomerance, C. and Rumely, R.S., 1983. On Distinguishing Prime Numbers from Composite Numbers. *Annals of Mathematics* [Online], 117(1), pp.173–206. Available from: `https://doi.org/10.2307%2F2006975`.

Agrawal, M., Kayal, N. and Saxena, N., 2004. PRIMES is in P. *Annals of Mathematics* [Online], 160(2), pp.781–793. Available from: `https://doi.org/10.4007/annals.2004.160.781`.

Atkin, A.O.L. and Morain, F., 1993. Elliptic Curves And Primality Proving. *Math. Comp*, 61, pp.29–68.

Baillie, R. and Wagstaff, S.S., 1980. Lucas pseudoprimes. *Mathematics of Computation* [Online], 35(152), pp.1391–1417. Available from: `https://doi.org/10.1090/S0025-5718-1980-0583518-6`.

Bernstein, D.J., 1998. Detecting perfect powers in essentially linear time. *Mathematics of Computation* [Online], 67(223), p.1253–1283. Available from: `https://doi.org/10.1090/S0025-5718-98-00952-1`.

Bernstein, D.J., 2003. *Proving Primality After Agrawal-Kayal-Saxena*. Unpublished.

Bernstein, D.J., 2004. *Distinguishing Prime Numbers from Composite Numbers: The State of the Art in 2004*. Unpublished.

Bernstein, D.J., 2007. Proving primality in essentially quartic random time. *Math. Comput.* [Online], 76(257), pp.389–403. Available from: `https://doi.org/10.1090/S0025-5718-06-01786-8`.

Bornemann, F., 2003. PRIMES is in P: A Breakthrough for "Everyman". *Notices of the American Mathematical Society* [Online], 50(5), pp.545–552. Available from: `http://www.ams.org/notices/200305/fea-bornemann.pdf`.

Bradford, R., 2020. CM30225: Parallel Computing Unit Notes [Online]. Available from: `https://people.bath.ac.uk/masrjb/CourseNotes/cm30225.html`.

Brent, R.P., 2010. Primality Testing.

Bronder, J., 2006. *The AKS Class of Primality Tests: A Proof of Correctness and Parallel Implementation* [Online]. Master's thesis. The University of Maine. Available from: `https://digitalcommons.library.umaine.edu/etd/1039`.

Caldwell, C., 2021. *The PrimePages: prime number research & records* [Online]. University of Tennessee at Martin. Available from: `https://primes.utm.edu/` [Accessed 2021-04-19].

Cheng, Q., 2007. Primality proving via one round in ECPP and one iteration in AKS. *Journal of Cryptology* [Online], 20(3), p.375–387. Available from: `https://doi.org/10.1007/s00145-006-0406-9`.

Cohen, H. and Lenstra, A.K., 1987. Implementation of a New Primality Test. *Mathematics of Computation* [Online], 48(177), pp.103–121. Available from: `http://www.jstor.org/stable/2007877`.

Cohen, H. and Lenstra, H.W., 1984. Primality testing and Jacobi sums. *Mathematics of Computation* [Online], 42(165), pp.297–330. Available from: `https://doi.org/10.2307%2F2007581`.

Cook, S.A., 1966. *On the minimum computation time of functions* [Online]. Ph.D. thesis. Department of Mathematics, Harvard University. Available from: `http://cr.yp.to/bib/1966/cook.html`.

Crandall, R. and Papadopoulos, J., 2003. *On the implementation of AKS-class primality tests*. University of Maryland College Park, Tech. Rep.

Crandall, R. and Pomerance, C., 2006. *Prime Numbers: A Computational Perspective.*, Lecture notes in statistics. Springer New York.

Davenport, H., 2008. *The Higher Arithmetic: An Introduction to the Theory of Numbers*. 8th ed. Cambridge: Cambridge University Press.

Davenport, J.H., 2021. *Computer Algebra* [Online]. Available from: `https://people.bath.ac.uk/masjhd/JHD-CA.pdf` [Accessed 2021-03-19].

Dietzfelbinger, M., 2004. *Primality Testing in Polynomial Time: From Randomized Algorithms to "PRIMES Is in P"* [Online]. 1st ed. Berlin, Heidelberg: Springer-Verlag. Available from: `https://doi.org/10.1007/b12334`.

Du, P., Weber, R., Luszczek, P., Tomov, S., Peterson, G. and Dongarra, J., 2012. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Computing* [Online], 38(8), pp.391–407. Available from: `https://doi.org/https://doi.org/10.1016/j.parco.2011.10.002`.

Egan, S., 2005. *Investigating Polynomial Time Primality Testing* [Online]. Master's thesis. Department of Computer Science, University of Bath, Bath, United Kingdom. Available from: `https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.463.1596&rep=rep1&type=pdf`.

Ewart, T., Hehn, A. and Troyer, M., 2013. Vli – a library for high precision integer and polynomial arithmetic. In: J.M. Kunkel, T. Ludwig and H.W. Meuer, eds. *Supercomputing*. Berlin, Heidelberg: Springer Berlin Heidelberg, pp.267–278.

Gallot, Y., 2005. *Implementation of the AKS algorithm* [Online]. Available from: `http://perso.wanadoo.fr/yves.gallot/index.html` [Accessed 2021-03-25].

Goldwasser, S. and Kilian, J., 1986. Almost All Primes Can Be Quickly Certified [Online]. *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*.

New York, NY, USA: Association for Computing Machinery, STOC '86, p.316–329. Available from: `https://doi.org/10.1145/12130.12162`.

Goldwasser, S. and Kilian, J., 1999. Primality Testing Using Elliptic Curves. *J. ACM* [Online], 46(4), p.450–472. Available from: `https://doi.org/10.1145/320211.320213`.

Granlund, T., 1991. *GMP: The GNU Multiple Precision Arithmetic Library* [Online]. Available from: `https://gmplib.org/` [Accessed 2021-04-19].

Granlund, T., 2020. *GNU MP 6.2.1: A GNU Manual* [Online]. Available from: `https://gmplib.org/manual/` [Accessed 2021-04-19].

Granville, A., 2004. It Is Easy to Determine Whether a Given Integer Is Prime. *Bulletin of the American Mathematical Society* [Online], 42(01), pp.3–39. Available from: `https://doi.org/10.1090/s0273-0979-04-01037-7`.

Haines, T.F., 2021. *Hex* [Online]. Bath, United Kingdom: Department of Computer Science, University of Bath. Available from: `https://hex.cs.bath.ac.uk/`.

Harris, M., 2004. *GPGPU: Beyond Graphics* [Online]. Available from: `http://download.nvidia.com/developer/presentations/GDC_2004/GDC2004_OpenGL_GPGPU_04.pdf` [Accessed 2021-04-19].

Harvey, D. and Hart, W., 2007. *FLINT: Fast Library for Number Theory* [Online]. Available from: `http://flintlib.org/` [Accessed 2021-08-19].

Harvey, D. and van der Hoeven, J., 2021. Integer multiplication in time $O(n \log n)$. *Annals of Mathematics* [Online], 193(2), p.563. Available from: `https://doi.org/10.4007/annals.2021.193.2.4`.

Joldes, M., Muller, J.M., Popescu, V. and Tucker, W., 2016. Campary: Cuda multiple precision arithmetic library and applications. In: G.M. Greuel, T. Koch, P. Paule and A. Sommese, eds. *Mathematical software – icms 2016*. Springer International Publishing, pp.232–240.

Karatsuba, A. and Ofman, Y., 1963. Multiplication of Multidigit Numbers on Automata. *Soviet Physics Doklady*, 7, pp.595–596.

Laird, J., 2020. CM30173: Cryptography Unit Notes.

Lelechenko, A. and Fischer, D., 2020. *arithmoi: Efficient basic number-theoretic functions* [Online]. Available from: `https://hackage.haskell.org/package/arithmoi` [Accessed 2021-03-25].

Lenstra, A.K. and Lenstra, H.W., 1991. *Algorithms in Number Theory*, Cambridge, MA, USA: MIT Press, p.673–715.

Lenstra, H.W., 1987. Factoring Integers with Elliptic Curves. *Annals of Mathematics* [Online], 126(3), pp.649–673. Available from: `http://www.jstor.org/stable/1971363`.

Lenstra, H.W., 2002. *Primality testing with cyclotomic rings*. Unpublished.

Lenstra, H.W. and Pomerance, C., 2019. Primality testing with Gaussian periods.

Journal of the European Mathematical Society [Online], 21(4), p.1229–1269. Available from: https://doi.org/10.4171/JEMS/861.

Li, H., 2007. *The analysis and implementation of the AKS algorithm and its improvement algorithms*. (Computer Science Technical Reports, CSBU-2007-09). Bath, United Kingdom: Department of Computer Science, University of Bath.

Menon, V., 2013. Deterministic Primality Testing - understanding the AKS algorithm [Online]. 1311.3785, Available from: https://arxiv.org/abs/1311.3785.

Miller, G.L., 1976. Riemann's hypothesis and tests for primality. *Journal of Computer and System Sciences* [Online], 13(3), pp.300–317. Available from: https://doi.org/https://doi.org/10.1016/S0022-0000(76)80043-8.

NVIDIA, 2007. *CUDA Toolkit* [Online]. Available from: https://developer.nvidia.com/cuda-toolkit [Accessed 2021-04-19].

NVIDIA, 2020. *GeForce RTX 3090 Specifications* [Online]. Available from: https://www.nvidia.com/en-gb/geforce/graphics-cards/30-series/rtx-3090/ [Accessed 2021-04-19].

NVIDIA, 2021a. *CUDA Toolkit Documentation - v11.3.0* [Online]. Available from: https://docs.nvidia.com/cuda/index.html [Accessed 2021-04-19].

NVIDIA, 2021b. *CUDA Toolkit Documentation: cuFFT - v11.3.0* [Online]. Available from: https://docs.nvidia.com/cuda/cufft/index.html [Accessed 2021-04-19].

NVIDIA, 2021c. *CUDA Toolkit FAQ* [Online]. Available from: https://developer.nvidia.com/cuda-faq [Accessed 2021-04-19].

Pomerance, C., 1984. Are there counter-examples to the Baillie – PSW primality test? [Online]. Available from: http://www.pseudoprime.com/pseudo.html.

Pomerance, C., Selfridge, J.L. and Wagstaff, S.S., 1980. The Pseudoprimes to 25×10^9. *Mathematics of Computation* [Online], 35(151), pp.1003–1026. Available from: https://doi.org/10.1090/S0025-5718-1980-0572872-7.

Powell, T. and Vorobjov, N., 2020. CM20217: Foundations of Computation Lecture Notes.

Rabin, M., 1980. Probabilistic algorithm for testing primality. *Journal of Number Theory*, 12, pp.128–138.

Rotella, C., 2005. *An Efficient Implementation of the AKS Polynomial-Time Primality Proving Algorithm* [Online]. Master's thesis. Carnegie Mellon University, Pittsburgh, PA. Available from: https://www.cs.cmu.edu/afs/cs/user/mjs/ftp/thesis-program/2005/rotella.pdf.

Ruderman, J., 2021. *Google C++ Style Guide* [Online]. Available from: https://google.github.io/styleguide/cppguide.html [Accessed 2021-07-17].

Salembier, R.G. and Southerington, P., 2005. An Implementation of the AKS Primality Test. *Computer Engineering*.

Schönhage, A. and Strassen, V., 1971. Schnelle Multiplikation großer Zahlen. *Computing* [Online], 7(3), pp.281–292. Available from: `https://doi.org/https://doi.org/10.1007/BF02242355`.

Scott, M., n.d. *MIRACL: Multiprecision Integer and Rational Arithmetic C/C++ Library* [Online]. Available from: `https://github.com/miracl/MIRACL` [Accessed 2021-04-19].

Shoup, V., 2001a. *A Tour of NTL* [Online]. Available from: `https://libntl.org/doc/tour.html` [Accessed 2021-04-21].

Shoup, V., 2001b. *NTL: A Library for doing Number Theory* [Online]. Available from: `https://libntl.org/` [Accessed 2021-04-19].

Solovay, R. and Strassen, V., 1977. A Fast Monte-Carlo Test for Primality. *SIAM Journal on Computing* [Online], 6(1), pp.84–85. `https://doi.org/10.1137/0206006`, Available from: `https://doi.org/10.1137/0206006`.

Takahashi, D., 2010. Parallel implementation of multiple-precision arithmetic and $2,576,$
$980,370,000$ decimal digits of π calculation. *Parallel Computing* [Online], 36, pp.439–448. Available from: `https://doi.org/10.1016/j.parco.2010.02.007`.

techpowerup.com, 2020. *GeForce RTX 3090 Specs* [Online]. Available from: `https://www.techpowerup.com/gpu-specs/geforce-rtx-3090.c3622` [Accessed 2021-03-19].

Toom, A.L., 1963. The complexity of a scheme of functional elements realizing the multiplication of integers. *Soviet Mathematics Doklady*, 3, pp.714–716.

Walton, J., 2020. *GeForce RTX 3090 Review* [Online]. Available from: `https://www.tomshardware.com/uk/reviews/nvidia-geforce-rtx-3090-review` [Accessed 2021-04-19].

About the Author

Vincent Renders is a bilingual (excluding programming languages) computer scientist and graduate aeronautical engineer from London. He completed his undergraduate degree at Loughborough University in 2020, where his prize-winning final year project on self-driving bicycles nurtured a curiosity for all things programming and computing.

Following the completion of his dissertation and his MSc. at Bath, Vincent looks forward to enjoying his time off and spending some time in business development with an e-commerce technology company before taking up his graduate position in Software Engineering at *Accenture UK* in March 2022.

When not behind a computer, tinkering with his latest *Arduino*, *Raspberry Pi*, or other DIY project, Vincent can be found racing bicycles or seeking other outdoor adventures somewhere between London and Cambridge.

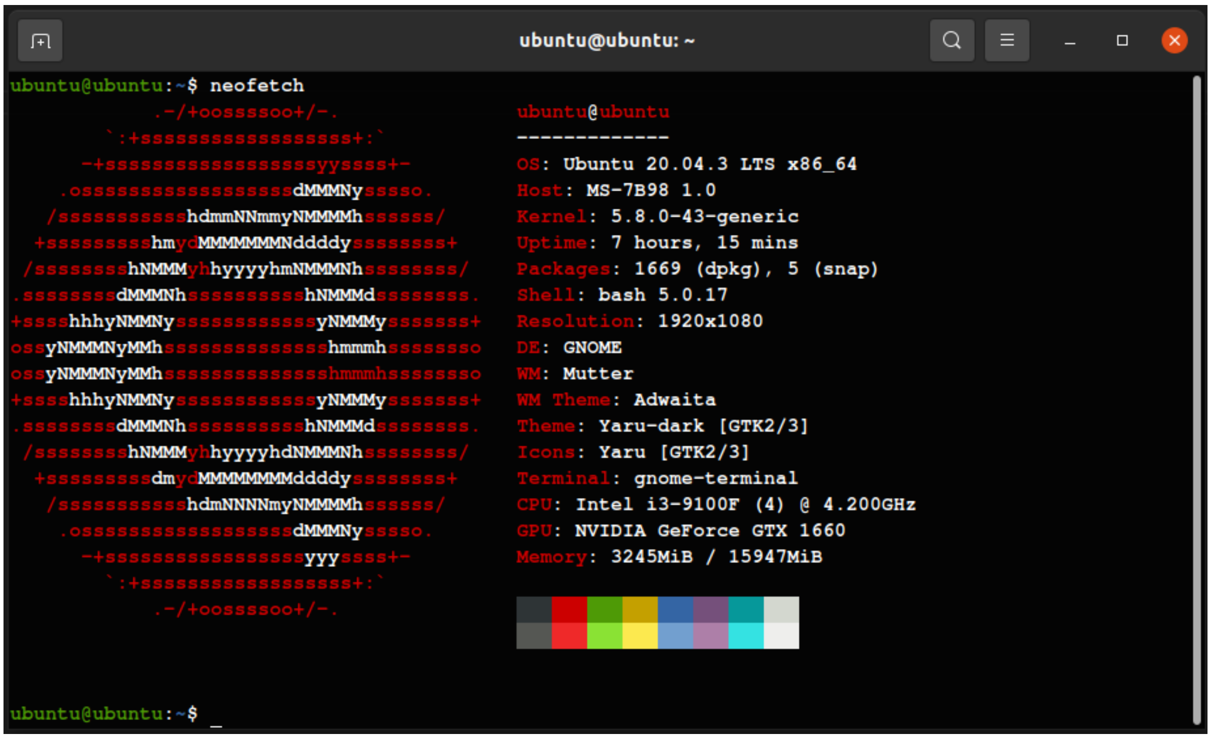
Vincent accepts correspondence via LinkedIn, GitHub, or email.

APPENDICES

Appendix A

Supplementary Material

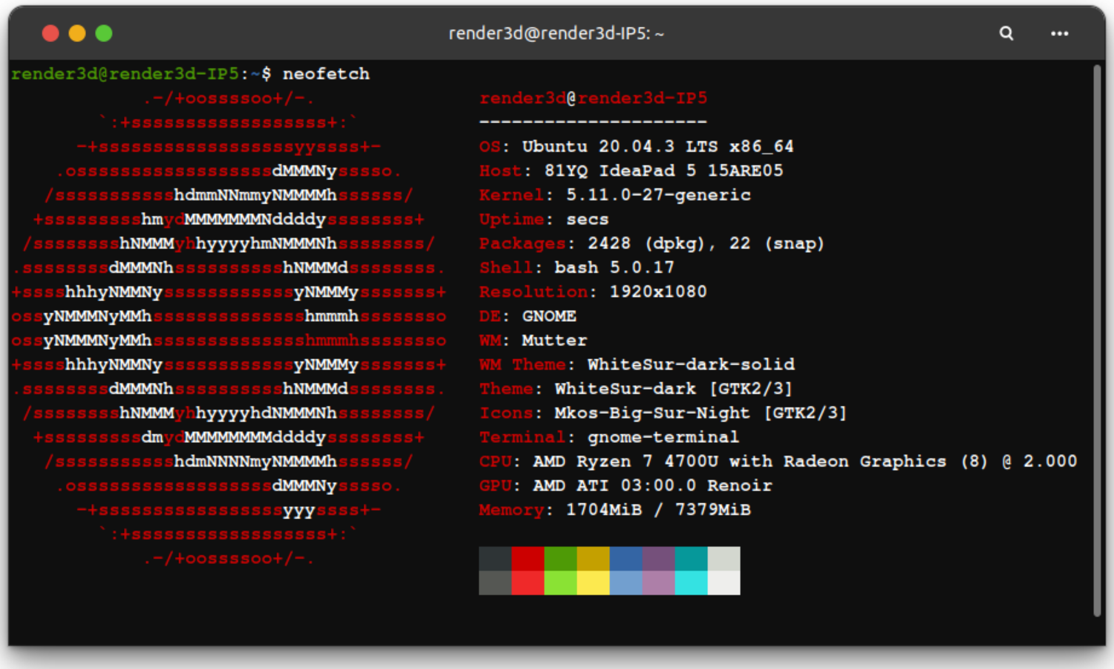
A.1 Machine Hardware Specifications



ubuntu@ubuntu: ~

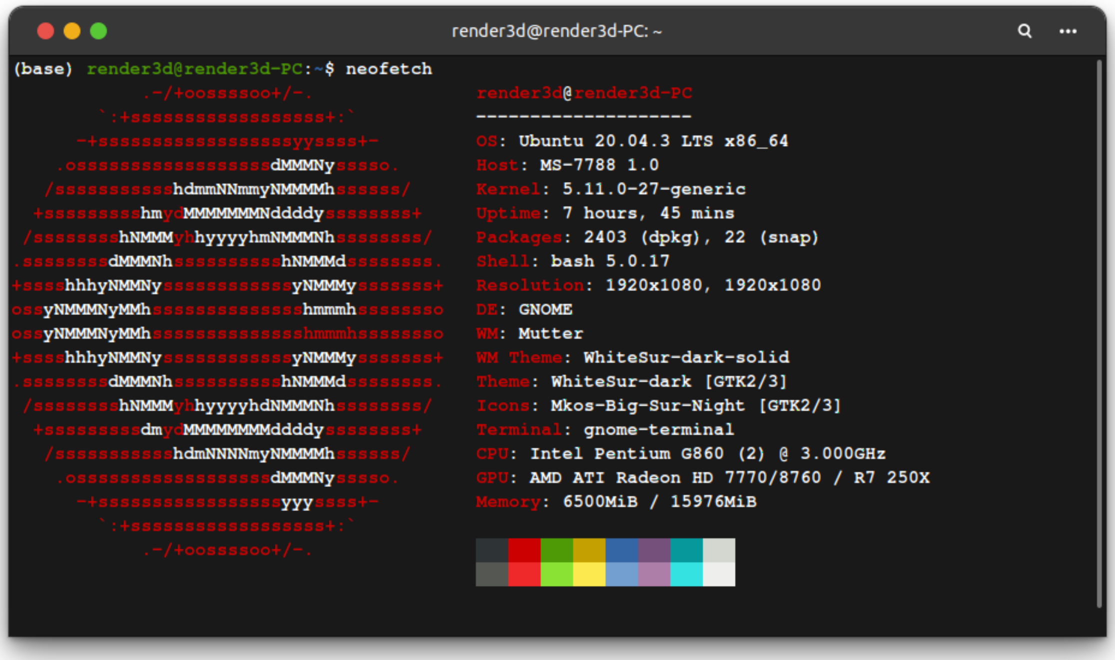
```
ubuntu@ubuntu:~$ neofetch
            .-/+oossssoo+/-.               ubuntu@ubuntu
        `:+ssssssssssssssssss+:`           -------------
      -+ssssssssssssssssssyyssss+-         OS: Ubuntu 20.04.3 LTS x86_64
    .ossssssssssssssssssdMMMNysssso.       Host: MS-7B98 1.0
   /ssssssssssshdmmNNmmyNMMMMhssssss/      Kernel: 5.8.0-43-generic
  +ssssssssshmydMMMMMMMNddddyssssssss+     Uptime: 7 hours, 15 mins
 /sssssssshNMMMyhhyyyyhmNMMMNhssssssss/    Packages: 1669 (dpkg), 5 (snap)
.ssssssssdMMMNhsssssssssshNMMMdssssssss.   Shell: bash 5.0.17
+sssshhhyNMMNyssssssssssssyNMMMysssssss+   Resolution: 1920x1080
ossyNMMMNyMMhsssssssssssssshmmmhssssssso   DE: GNOME
ossyNMMMNyMMhsssssssssssssshmmmhssssssso   WM: Mutter
+sssshhhyNMMNyssssssssssssyNMMMysssssss+   WM Theme: Adwaita
.ssssssssdMMMNhsssssssssshNMMMdssssssss.   Theme: Yaru-dark [GTK2/3]
 /sssssssshNMMMyhhyyyyhdNMMMNhssssssss/    Icons: Yaru [GTK2/3]
  +sssssssssdmydMMMMMMMMddddyssssssss+     Terminal: gnome-terminal
   /ssssssssssshdmNNNNmyNMMMMhssssss/      CPU: Intel i3-9100F (4) @ 4.200GHz
    .ossssssssssssssssssdMMMNysssso.       GPU: NVIDIA GeForce GTX 1660
      -+sssssssssssssssssyyyssss+-         Memory: 3245MiB / 15947MiB
        `:+ssssssssssssssssss+:`
            .-/+oossssoo+/-.


ubuntu@ubuntu:~$ _
```

Figure A.1: The hardware specifications for the main computer, a desktop PC from *c.* 2019, used for conducting profiling tests and running AKS in serial and parallel. Note the dedicated *NVIDIA* GPU and quad-core *Intel* processor.



```
render3d@render3d-IP5:~$ neofetch
            .-/+oossssoo+/-.               render3d@render3d-IP5
        `:+ssssssssssssssssss+:`           ---------------------
      -+ssssssssssssssssssyyssss+-         OS: Ubuntu 20.04.3 LTS x86_64
    .ossssssssssssssssssdMMMNysssso.       Host: 81YQ IdeaPad 5 15ARE05
   /ssssssssssshdmmNNmmyNMMMMhssssss/      Kernel: 5.11.0-27-generic
  +ssssssssshmydMMMMMMMNddddyssssssss+     Uptime: secs
 /sssssssshNMMMyhhyyyyhmNMMMNhssssssss/    Packages: 2428 (dpkg), 22 (snap)
.ssssssssdMMMNhsssssssssshNMMMdssssssss.   Shell: bash 5.0.17
+sssshhhyNMMNyssssssssssssyNMMMysssssss+   Resolution: 1920x1080
ossyNMMMNyMMhsssssssssssssshmmmhssssssso   DE: GNOME
ossyNMMMNyMMhsssssssssssssshmmmhssssssso   WM: Mutter
+sssshhhyNMMNyssssssssssssyNMMMysssssss+   WM Theme: WhiteSur-dark-solid
.ssssssssdMMMNhsssssssssshNMMMdssssssss.   Theme: WhiteSur-dark [GTK2/3]
 /sssssssshNMMMyhhyyyyhdNMMMNhssssssss/    Icons: Mkos-Big-Sur-Night [GTK2/3]
  +sssssssssdmydMMMMMMMMddddyssssssss+     Terminal: gnome-terminal
   /ssssssssssshdmNNNNmyNMMMMhssssss/      CPU: AMD Ryzen 7 4700U with Radeon Graphics (8) @ 2.000
    .ossssssssssssssssssdMMMNysssso.       GPU: AMD ATI 03:00.0 Renoir
      -+sssssssssssssssssyyyssss+-         Memory: 1704MiB / 7379MiB
        `:+ssssssssssssssssss+:`
            .-/+oossssoo+/-.
```

Figure A.2: The hardware specifications for the secondary computer, a laptop from *c.* 2021, used primarily for development work and running AKS in serial and parallel. Note the eight-core *AMD* processor.



```
(base) render3d@render3d-PC:~$ neofetch
            .-/+oossssoo+/-.               render3d@render3d-PC
        `:+ssssssssssssssssss+:`           --------------------
      -+ssssssssssssssssssyyssss+-         OS: Ubuntu 20.04.3 LTS x86_64
    .ossssssssssssssssssdMMMNysssso.       Host: MS-7788 1.0
   /ssssssssssshdmmNNmmyNMMMMhssssss/      Kernel: 5.11.0-27-generic
  +ssssssssshmydMMMMMMMNddddyssssssss+     Uptime: 7 hours, 45 mins
 /sssssssshNMMMyhhyyyyhmNMMMNhssssssss/    Packages: 2403 (dpkg), 22 (snap)
.ssssssssdMMMNhsssssssssshNMMMdssssssss.   Shell: bash 5.0.17
+sssshhhyNMMNyssssssssssssyNMMMysssssss+   Resolution: 1920x1080, 1920x1080
ossyNMMMNyMMhsssssssssssssshmmmhssssssso   DE: GNOME
ossyNMMMNyMMhsssssssssssssshmmmhssssssso   WM: Mutter
+sssshhhyNMMNyssssssssssssyNMMMysssssss+   WM Theme: WhiteSur-dark-solid
.ssssssssdMMMNhsssssssssshNMMMdssssssss.   Theme: WhiteSur-dark [GTK2/3]
 /sssssssshNMMMyhhyyyyhdNMMMNhssssssss/    Icons: Mkos-Big-Sur-Night [GTK2/3]
  +sssssssssdmydMMMMMMMMddddyssssssss+     Terminal: gnome-terminal
   /ssssssssssshdmNNNNmyNMMMMhssssss/      CPU: Intel Pentium G860 (2) @ 3.000GHz
    .ossssssssssssssssssdMMMNysssso.       GPU: AMD ATI Radeon HD 7770/8760 / R7 250X
      -+sssssssssssssssssyyyssss+-         Memory: 6500MiB / 15976MiB
        `:+ssssssssssssssssss+:`
            .-/+oossssoo+/-.
```

Figure A.3: The hardware specifications for the backup computer, a desktop PC from *c.* 2012, used primarily for development and testing of programs. Note this difference between the hardware on this computer; compare with that of the other two and take a moment to consider the developments in computer hardware over the last decade.

A.2 Comparing the Carmichael Lambda and Euler Phi Functions

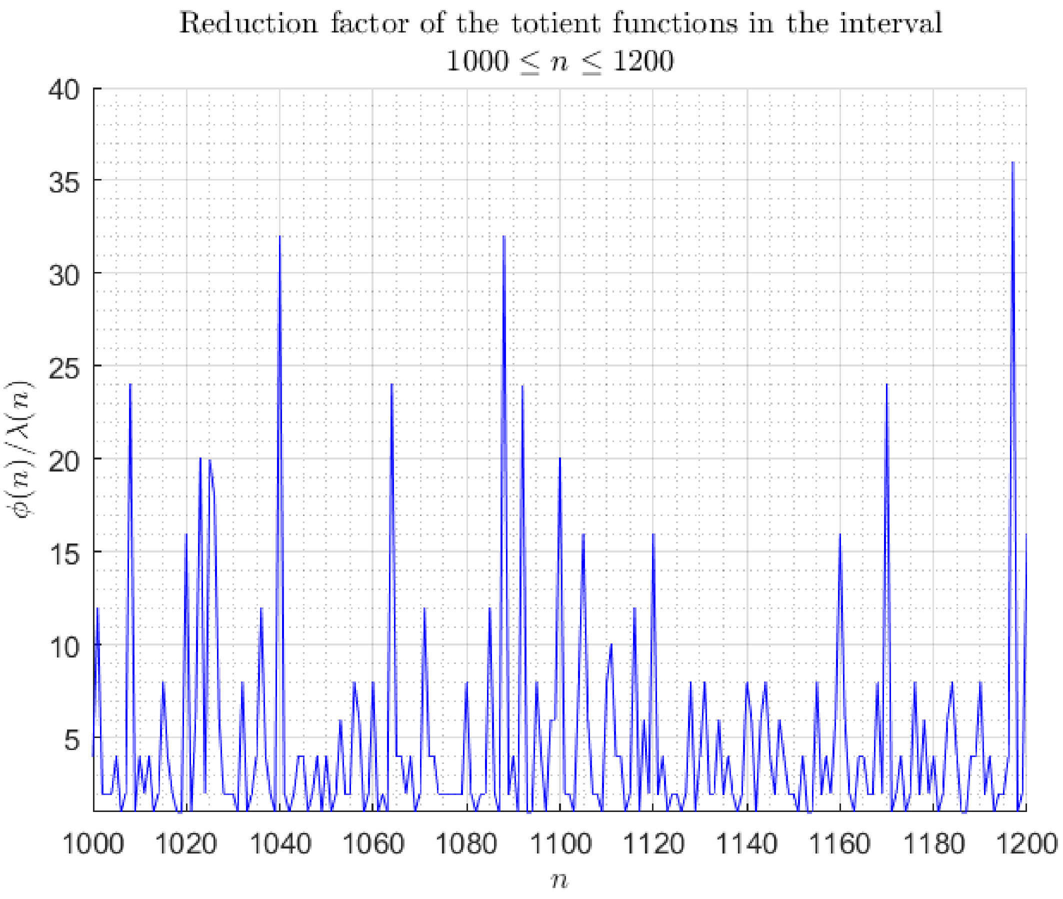


Figure A.4: The factor by which a is reduced when the Carmichael lambda function is used in place of the Euler totient for relatively small values of n.

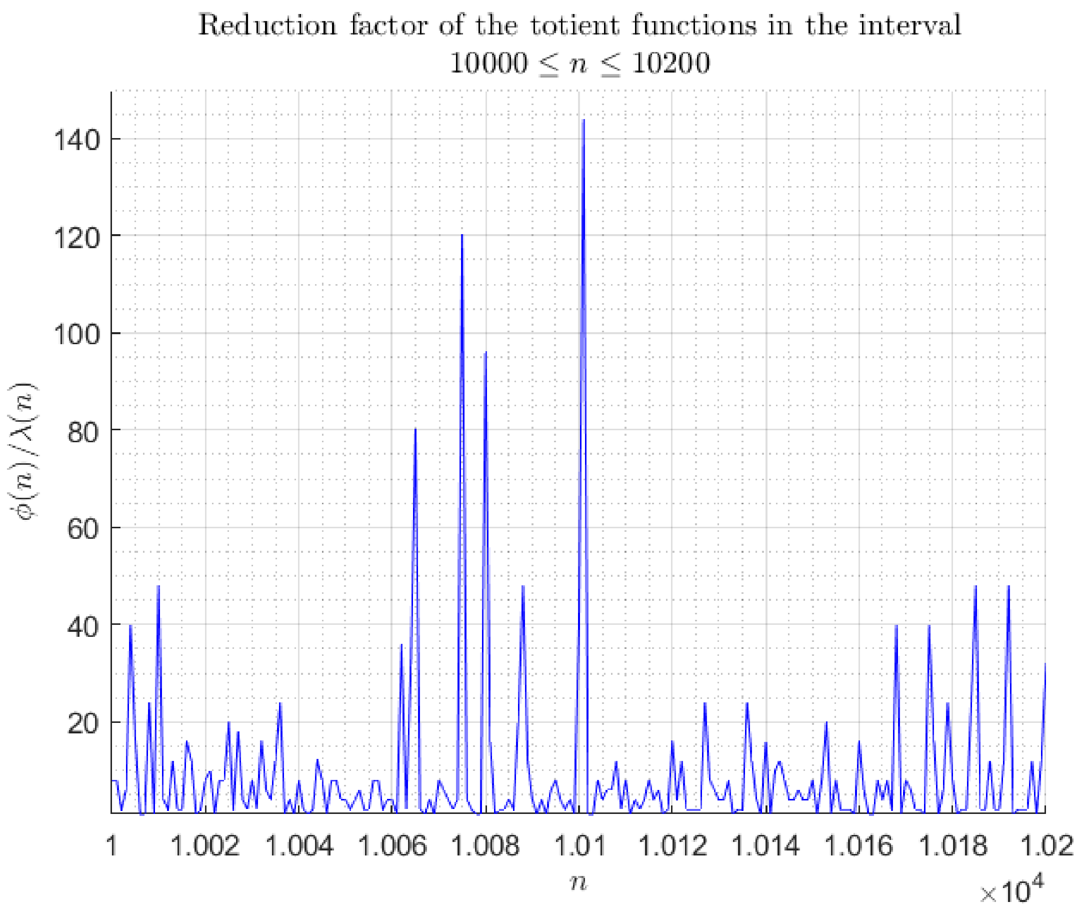


Figure A.5: The factor by which a is reduced when the Carmichael lambda function is used in place of the Euler totient for larger values of n.

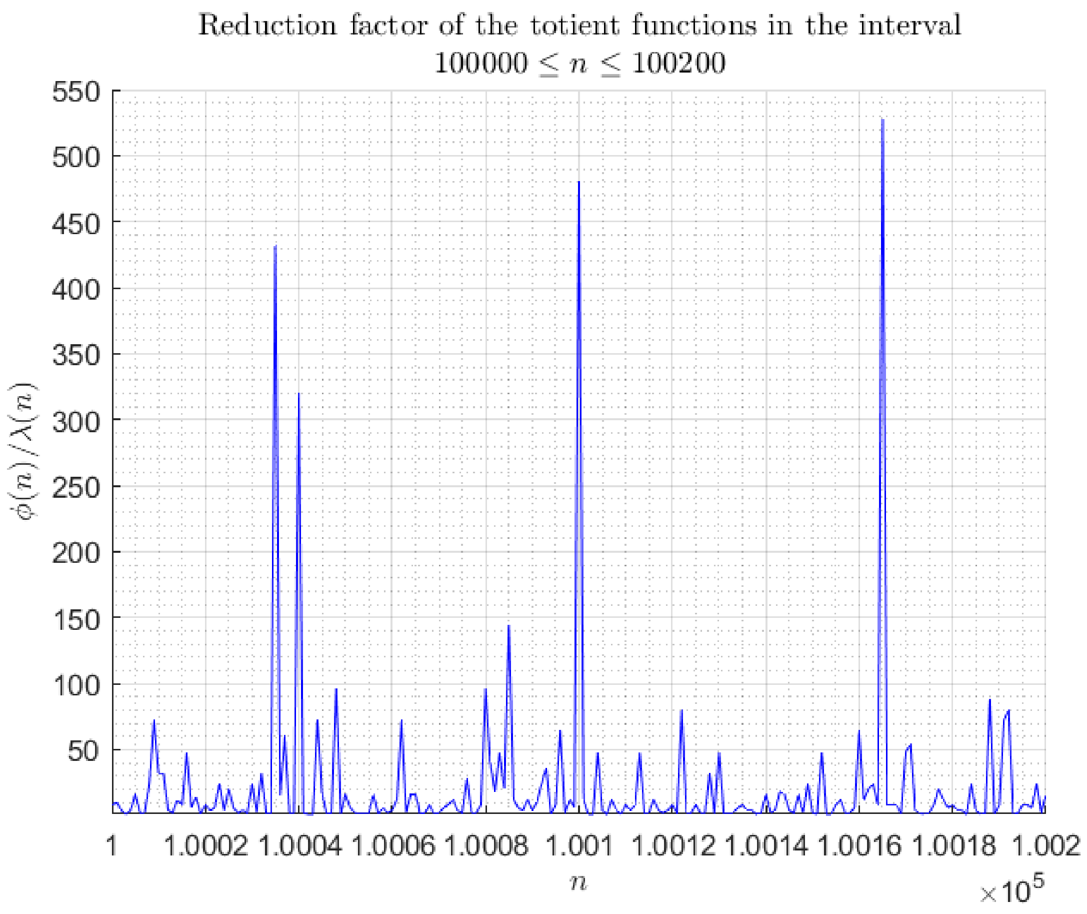


Figure A.6: The factor by which a is reduced when the Carmichael lambda function is used in place of the Euler totient for large values of n. The phenomenon persists even to numbers of this size. Note how a significant reduction factor is likely to yield equally significant runtime savings if $\lambda(r)$ is used in place of $\phi(r)$ when calculating a.

A.3 Additional Data

Distribution of the values of r for the primes $p \in \{n \mid N - 210 \leq n \leq N\}$,
where $N = 5343707999999999999999994656293$

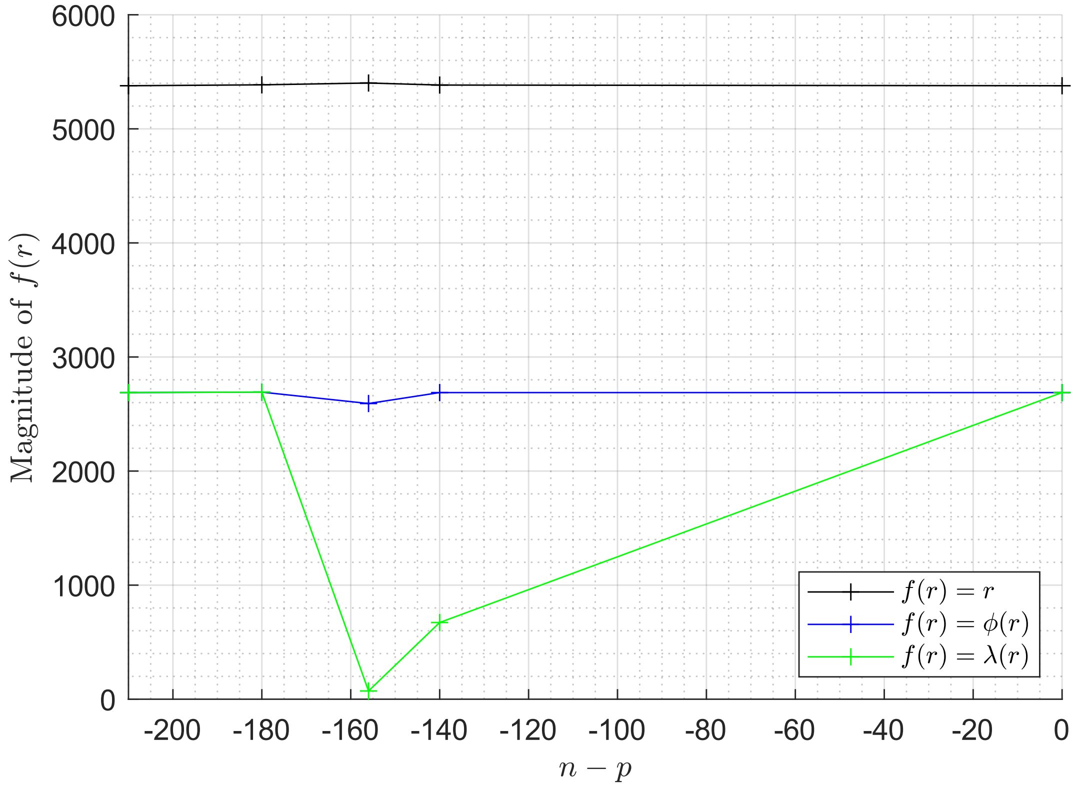


Figure A.7

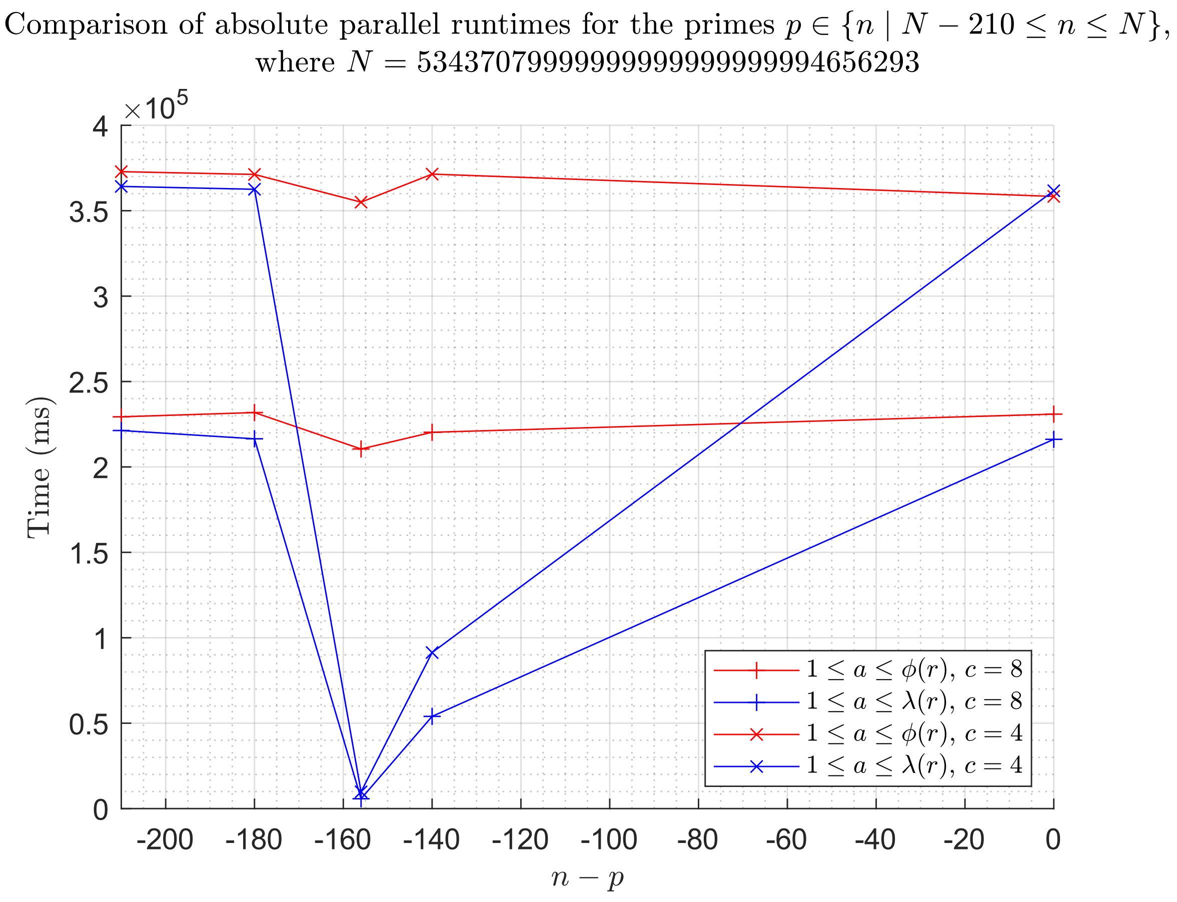
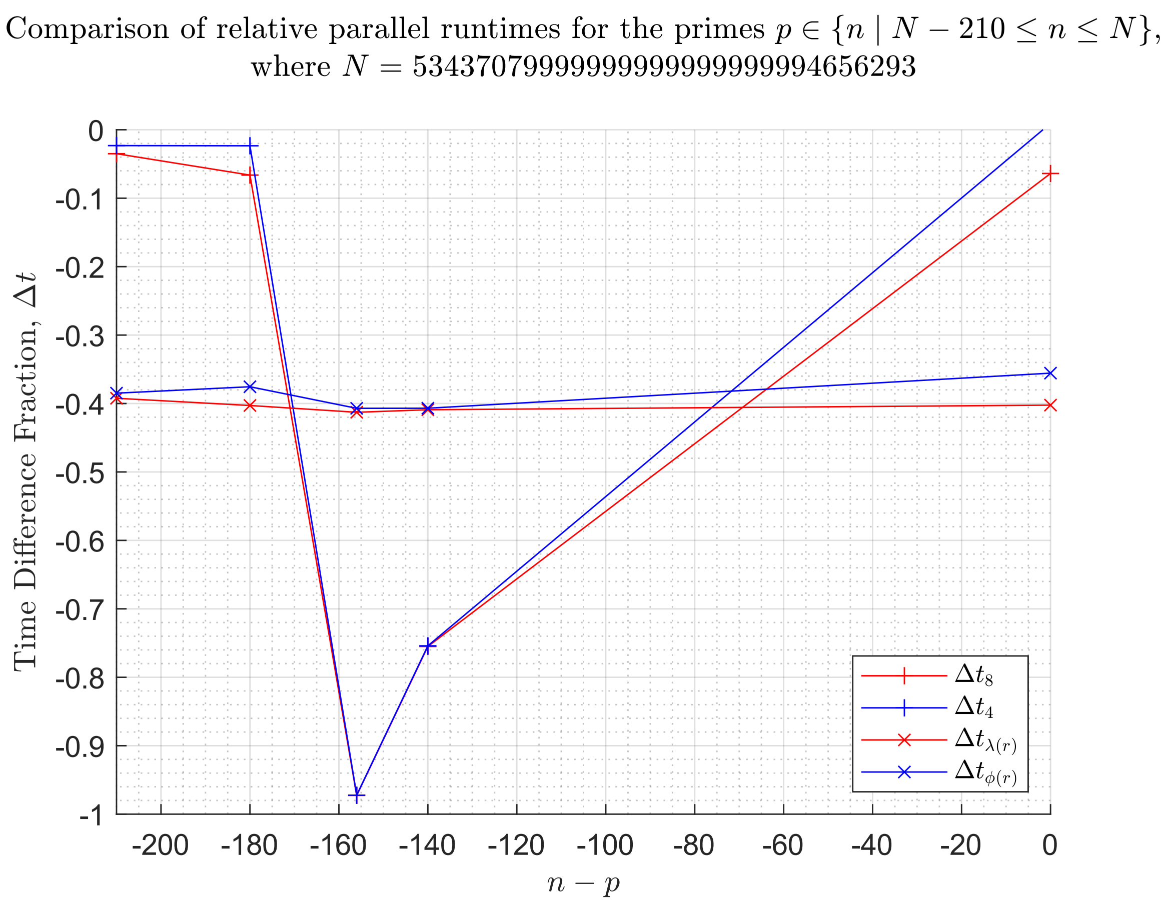
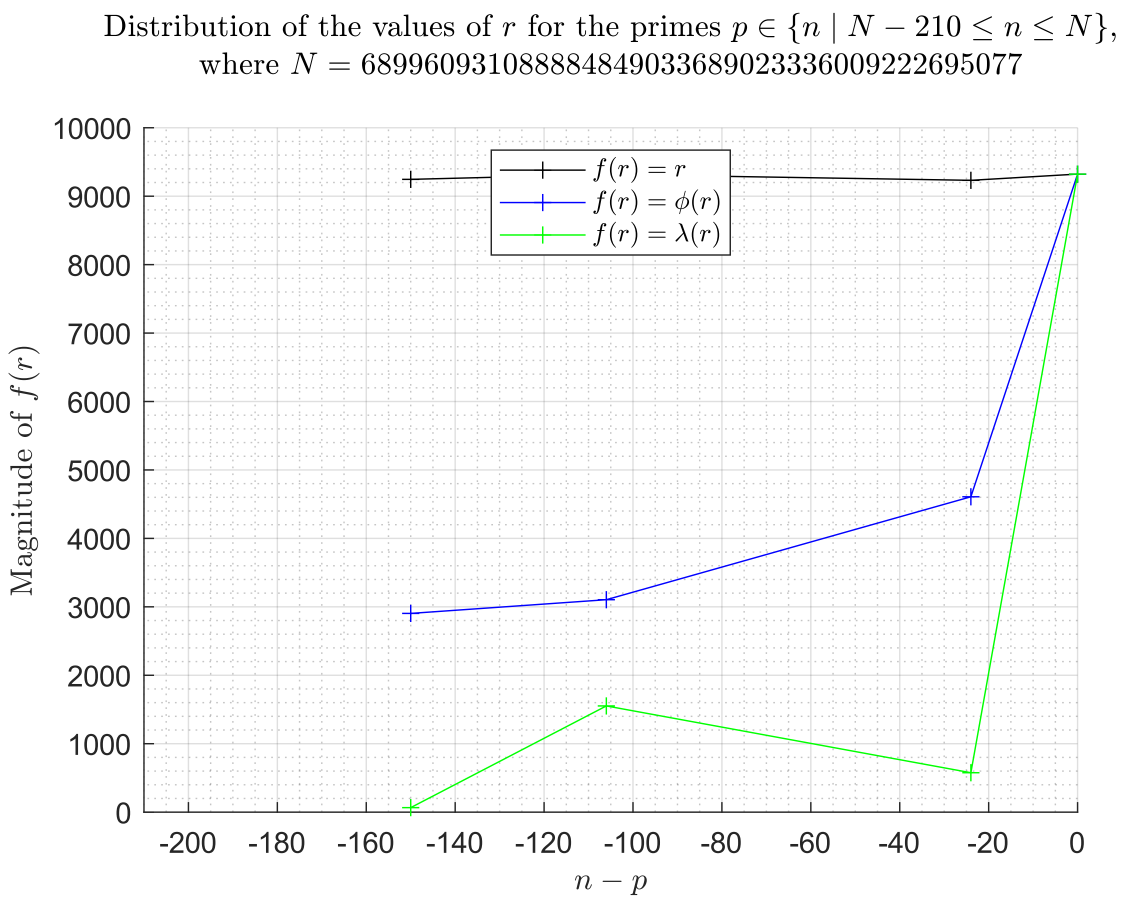

Comparison of absolute parallel runtimes for the primes $p \in \{n \mid N - 210 \leq n \leq N\}$,
where $N = 53437079999999999999999994656293$
$\times 10^5$
4
3.5
3
2.5
2
1.5
1
0.5
0
Time (ms)
-200
-180
-160
-140
-120
-100
-80
-60
-40
-20
0
$n - p$
$1 \leq a \leq \phi(r)$, $c = 8$
$1 \leq a \leq \lambda(r)$, $c = 8$
$1 \leq a \leq \phi(r)$, $c = 4$
$1 \leq a \leq \lambda(r)$, $c = 4$

Figure A.8



Comparison of relative parallel runtimes for the primes $p \in \{n \mid N - 210 \leq n \leq N\}$,
where $N = 534370799999999999999999994656293$
Time Difference Fraction, Δt
$n - p$
0
-0.1
-0.2
-0.3
-0.4
-0.5
-0.6
-0.7
-0.8
-0.9
-1
-200
-180
-160
-140
-120
-100
-80
-60
-40
-20
0
Δt_8
Δt_4
$\Delta t_{\lambda(r)}$
$\Delta t_{\phi(r)}$

Figure A.9



Distribution of the values of r for the primes $p \in \{n \mid N - 210 \leq n \leq N\}$,
where $N = 6899609310888848490336890233360092226950 77$
$f(r) = r$
$f(r) = \phi(r)$
$f(r) = \lambda(r)$
Magnitude of $f(r)$
$n - p$
10000
9000
8000
7000
6000
5000
4000
3000
2000
1000
0
-200
-180
-160
-140
-120
-100
-80
-60
-40
-20
0

Figure A.10

Comparison of absolute parallel runtimes for the primes $p \in \{n \mid N - 210 \leq n \leq N\}$, where $N = 68996093108888484903368902333600922269507 7$

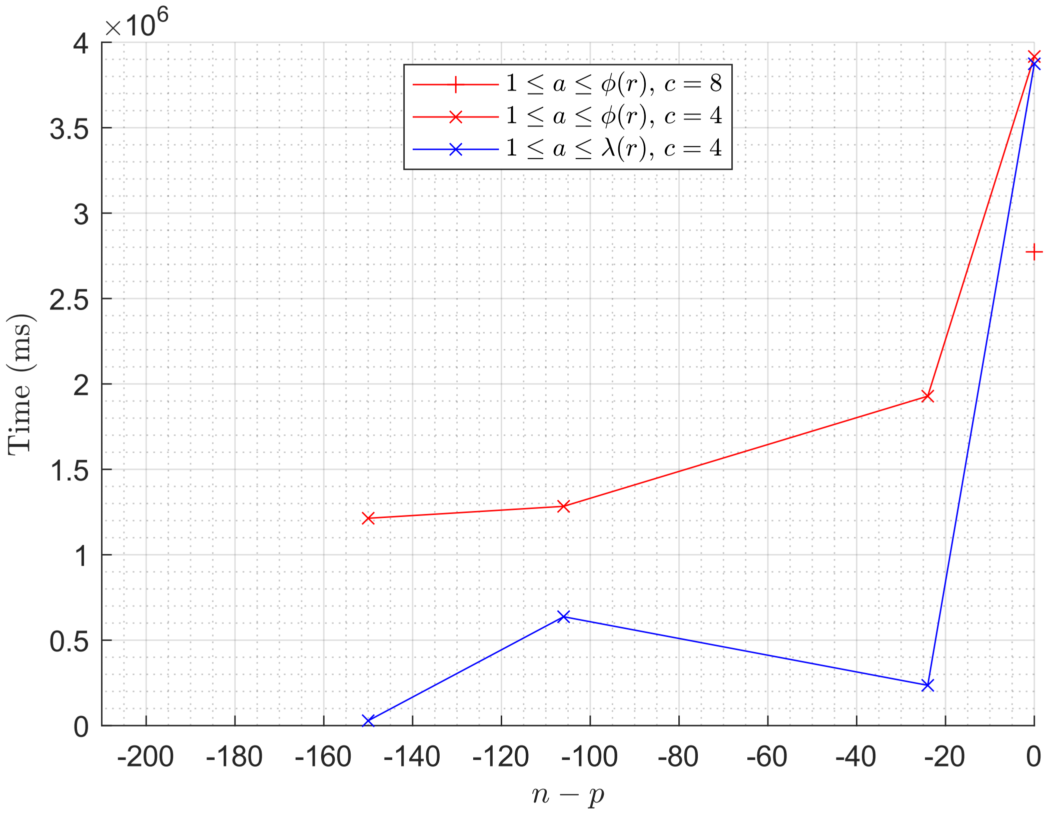


Figure A.11

Comparison of relative parallel runtimes for the primes $p \in \{n \mid N - 210 \leq n \leq N\}$, where $N = 689960931088884849033689023336009222695077$

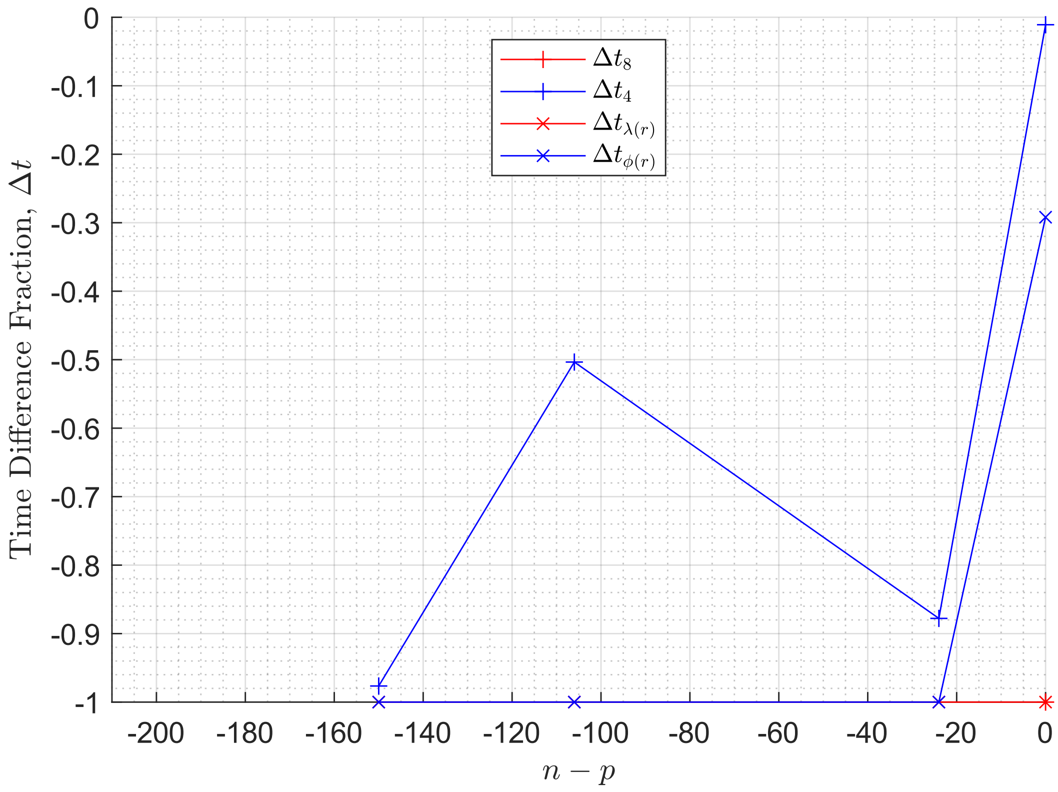


Figure A.12

Appendix B

List of Primes

B.1 Primes from Gallot (2005)

347

11701

23456789

1000000007

9007199254740997

B.2 Primes from Li (2007)

10133

32561

160583

268069

2969023

7741009

12434839

86457079

100982933

879673117

2492813497

435465768733

7000000000009

10000000000037

100000000000031

88903

99397

107339

777619

1000099

4740623

26933611

10012333

100041703

103736293

1001772091

2958347047

10015571677

44426255143

100020817331

333267326767

1000528294943

3222583708567

10083087720779

35466059872651

112272535095293

281702565146179

1003026954441971

4467165232203397

10022390619214807

100055128505716009

1083717775299973771

2954636327037869707

32607078403577476797

497300494190239610272

7510058567761205856771 03

614783152143098270145397

7666569009190249923345281

86084043198752959566539209

912613825844053990694091143

1000474617637553175973957663

20476096752860587951845236929

387121083116233373653498534849

4313339400115792413779939218099

53437079999999999999999994656293

617594269939999999993824057300601

7433112517127371273712736384060023

81156673714614395518740388458373409

974002888812604462999840811965244329

10409245707577777777777777673685320703

274910905008600000000000002749109050087

3570356436000600000000000035703564360007

41602775436631511111111110695083356744797

55132856630488887841876472900339410613059

689960931088884849033689023336009222695077

B.3 Primes from Bronder (2006)

104729

200000033

9999954997

Appendix C

Source Code

C.1 Documentation

A selection of programs and their headers relevant to the content of this dissertation. All files related to this project, including those included in this appendix, may also be found in the GitHub repository.

> **Disclaimer**: Code included in this appendix accurately reflects the state of files of the same name in the GitHub repository as of 20:00 on Friday 3rd September 2021. After this date, files may have been modified by the owner of the repository at their discretion, and the reproduction of them in this appendix may no longer be accurate.

The repository includes compressed source code of both NTL-11.5.1 (released on 23/06/2021) and GMP-6.2.0 (released on 18/01/2020), which were the versions used throughout this project. Although there are newer versions of GMP (6.2.1, released on 14/11/2020), a compatibility clash or version mismatch occurs when this version of NTL (11.5.1) is compiled from source with GMP-6.2.1 (or later) already installed on the target machine.

For the compiler `include` path to find the libraries automatically, both should be installed in the default directory (`/usr/local`). A complete guide to compiling NTL from source and installing with GMP is documented separately and will not be reproduced here.

This repository is intended to work with *Unix*-based operating systems only. The owner of the repository makes no apology for this as this is by far the most convenient development setup for any person(s) intending to install and make best use of NTL or GMP. For the readers sanity, the repository owner would advise against attempting to port this work to a *Windows* development environment in the strongest possible terms.

Finally, assuming all of the above is in order, all main program files (`.cpp`) and headers (`.h`) will compile with the following terminal or `bash` shell command: `g++ -g -O2 -std=c++11 -pthread -march=native source_directory/source_filename.cpp -o custom_directory/custom_filename.out -lntl -lgmp -lm`

C.2 The *Zn*(*x*) Version

C.2.1 LenstraZnx.cpp

Listing C.1: The main C++ source file for the $Z_{n(x)}$ version of AKS. This implementation has been adapted from the work of Li (2007), with some inspiration from Gallot (2005), and is currently configured for use of the Carmichael lambda function and running congruence test in parallel. This functionality can be removed or swapped by commenting out the relevant lines.

```

/*
    Compile with:
        $ g++ -g -O2 -std=c++11 -pthread -march=native devVR/
           LenstraZnx.cpp -o devVR/LenstraZnx.out -lntl -lgmp -lm
    Run with:
        $ ./devVR/LenstraZnx.out
*/

#include <math.h> // standard libraries
#include <fstream>
#include <iostream>
#include <sstream>
#include <iomanip>
// #include <windows.h>
#include <unistd.h>
#include <stdio.h>
#include <cstdio>
#include <stdlib.h>
#include <cstdlib>
#include <filesystem>
// #include <mmsystem.h>
#include <time.h>
#include <ctime>
#include <chrono>
#include <string>
#include <thread>
#include <array>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/stat.h>
#include <sys/types.h>

#include "NTL/ZZ.h" // NTL Libraries
#include "NTL/ZZ_p.h"
#include "NTL/ZZ_pX.h"
#include "NTL/ZZX.h"
#include "NTL/vec_ZZ.h"
#include <NTL/ZZ_pXFactoring.h>
#include <NTL/BasicThreadPool.h>
NTL_CLIENT

#include "PerfectPower.h" // Personal Headers
```

```
#include "Euler.h"
#include "Carmichael.h"
#include "CongruenceZnx.h"
#include "CongruenceZnxHPC.h"

std::string getDate() {
    auto t = std::time(nullptr);
    auto tm = *std::localtime(&t);

    std::ostringstream oss;
    oss << std::put_time(&tm, "%Y-%m-%d");
    auto date = oss.str();

    return date;
}

std::string getTime() {
    auto t = std::time(nullptr);
    auto tm = *std::localtime(&t);

    std::ostringstream oss;
    oss << std::put_time(&tm, "%H-%M-%S");
    auto time = oss.str();

    return time;
}

std::string getDateTime() {
    auto t = std::time(nullptr);
    auto tm = *std::localtime(&t);

    std::ostringstream oss;
    oss << std::put_time(&tm, "%Y-%m-%d-%H-%M-%S");
    auto datetime = oss.str();

    return datetime;
}

std::string getFilename() {
    std::string fldr = "logs/Znx/";
    std::string prfx = "LnstrZnx-";
    std::string sffx = getDateTime();
    std::string extn = ".csv";

    std::string filename = fldr + prfx + sffx + extn;

    return filename;
}

std::string filename = getFilename();
std::ofstream perflog(filename, std::ios::app); // output result
```

```
   into file

inline void fileWrite(const ZZ& n, const unsigned int& cores, const
   bool& PRIME, const long& time, const std::string& other) {
    perflog << n << "," << cores << "," << PRIME  << "," << time <<
       "," << other << "\n";
}

inline bool Lenstra (const ZZ& n) {
    if(n < 1){
        std::printf("Integer n needs to be positive.\n");
        return false;
    }
    else if(n == 1){
        std::printf("1 is neither prime or composite.\n");
        return false;
    }
    else if(n == 2){
        std::printf("2 is prime.\n");
        return true;
    }
    else if(n == 3){
        std::printf("3 is prime.\n");
        return true;
    }

    std::cout << "n = " << n << "\n\n";

    // start timing
    auto start = std::chrono::high_resolution_clock::now();

    // Test if n is a perfect power
    int PP = PerfectPower(n);

    // returns 1 if n is a perfect power, 0 otherwise;
    if(PP == 1){
        auto finish = std::chrono::high_resolution_clock::now();
        auto duration = finish - start;
        auto time = std::chrono::duration_cast<std::chrono::
           milliseconds>(duration).count();

        std::printf("%ld is not prime.\n",to_long(n));
        std::printf("%ld is a perfect power.\n",to_long(n));
        std::printf("Time taken: %ld milliseconds\n",time);

        std::string note = std::to_string(to_long(n)) + " is a
           perfect power";
        fileWrite(n,ncores,false,time,note);

        return false;
    }

    // Find a suitable r
    ZZ r = to_ZZ(2);
    ZZ R;
    ZZ r1;

    while(r < n){
        ZZ R = GCD(r, n);
        if(R != 1 ){
            auto finish = std::chrono::high_resolution_clock::now();
            auto duration = finish - start;
            auto time = std::chrono::duration_cast<std::chrono::
               milliseconds>(duration).count();

            std::printf("%ld is not prime.\n",to_long(n));
            std::printf("%ld is a divisor.\n",to_long(R));
            std::printf("Time taken: %ld milliseconds\n",time);

            std::string note = std::to_string(to_long(R)) + " is a
               divisor";
            fileWrite(n,ncores,false,time,note);

            return false;
            break;
        }
        else {
            ZZ v = to_ZZ(floor(power_long(to_long(log(n)), 2)));

            // order of n mod r is bigger than v;
            int p = 0;
            ZZ_p::init(r); // calculate mod r

            while(v <= r){
                ZZ x = to_ZZ(power_long(to_long(n), to_long(v))); //
                    calculates x = n^v
                ZZ_p z = to_ZZ_p(x);
                if(z == to_ZZ_p(1)){
                    r1 = r; // store value of r
                    r = n + 1;
                    break;
                }
                else{
                    v = v + 1;
                }
            }
        }
        r = r + 1;
    }

    r = r1;
    std::printf("r = %ld\n",to_long(r));
    // ZZ r2 = Euler(to_long(r));
    // std::printf("Phi(%ld) = %ld\n",to_long(r),to_long(r2));
    ZZ r2 = Carmichael(to_long(r));
    std::printf("Lambda(%ld) = %ld\n",to_long(r),to_long(r2));

    long a = to_long(r2 - 1);
    // long f = CongruenceZnx(n,r,r2,a);
    long f = CongruenceZnxHPC(n,r,r2,a);

    if(f == 0){
        auto finish = std::chrono::high_resolution_clock::now();
        auto duration = finish - start;
        auto time = std::chrono::duration_cast<std::chrono::
           milliseconds>(duration).count();

        std::printf("%ld is prime.\n",to_long(n));
        std::printf("Time taken: %ld milliseconds\n",time);

        // std::string note = "a = " + std::to_string(a) + "; End: a
            = " + std::to_string(f) + "; r = " + std::to_string(
           to_long(r)) + "; phi(r) = " + std::to_string(to_long(r2))
           ;
        std::string note = "a = " + std::to_string(a) + "; End: a =
           " + std::to_string(f) + "; r = " + std::to_string(to_long
           (r)) + "; lambda(r) = " + std::to_string(to_long(r2));
        fileWrite(n,ncores,true,time,note);

        return true;
    }
    else {
        auto finish = std::chrono::high_resolution_clock::now();
        auto duration = finish - start;
        auto time = std::chrono::duration_cast<std::chrono::
           milliseconds>(duration).count();

        std::printf("%ld is not prime.\n",to_long(n));
        std::printf("The a which fails is %ld\n",f);
        std::printf("Time taken: %ld milliseconds\n",time);

        // std::string note = "a = " + std::to_string(a) + "; End: a
            = " + std::to_string(f) + "; r = " + std::to_string(
           to_long(r)) + "; phi(r) = " + std::to_string(to_long(r2))
           ;
        std::string note = "a = " + std::to_string(a) + "; End: a =
           " + std::to_string(f) + "; r = " + std::to_string(to_long
           (r)) + "; lambda(r) = " + std::to_string(to_long(r2));
        fileWrite(n,ncores,false,time,note);

        return false;
    }
```

```

}

int main (int argc, char * argv[]) {

    perflog << "Int, Cores, Prime (T/F), Time (milliseconds),
       Comments\n";

    ZZ p = conv<ZZ>("23456789");

    bool prime = Lenstra(p);

}
```

C.2.2 CongruenceZnx.h

Listing C.2: The congruence testing header associated with the $Z_{n(x)}$ version of AKS. This function has been optimised for run time performance and uses the NTL routines for polynomial multiplication.

```

#include "NTL/ZZ_p.h"
#include "NTL/ZZ_pX.h"
NTL_CLIENT

unsigned int ncores = std::thread::hardware_concurrency(); //
   machine cores - may return 0 when not able to detect

long CongruenceZnx(const ZZ& n, const ZZ& r, const ZZ& r2, const
   long& a){
    // congruence test of polynomials in regular form

    ZZ_p::init(n);                        // initialise mod n

    printf("\nCalculating x^r - 1 (mod n) ...\n");
    ZZ_pX b = ZZ_pX(to_long(r), 1) - 1; // b = x^r - 1 (mod n);
    printf("Done.\n");

    printf("Initialising x (mod n) ...\n");
    ZZ_pX e = ZZ_pX(1, 1);                // e = x (mod n)
    printf("Done.\n");

    printf("Calculating x^n (mod (x^r - 1), n) ...\n");
    ZZ_pX d = PowerMod(e, n, b);          // d = x^n (mod b, n)
    printf("Done.\n");

    printf("\nCommencing congruence tests:\n\n");
    for(long j = 1; j <= a; ++j){
        printf("a = %ld\n",j);

        ZZ_pX c = ZZ_pX(1, 1) - j ;         // c = x - a (mod n);
        ZZ_pX f = PowerMod(c, n, b);        // f = (x - a)^n (mod b,
            n) - LHS
        ZZ_pX g = d - j;                    // g = x^n - a (mod b, n
           ) - RHS

        if(f != g){
            return(j); // n is not prime
        }
    }

    return(0); // n is prime
}
```

C.3 The *Z* Version

C.3.1 LenstraZ.cpp

Listing C.3: The main C++ source file for the *Z* version of AKS. This program is almost identical to `LenstraZnx.cpp` save for the different `CongruenceZ` function call.

```

/*
    Compile with:
        $ g++ -g -O2 -std=c++11 -pthread -march=native devVR/
           LenstraZ.cpp -o devVR/LenstraZ.out -lntl -lgmp -lm
    Run with:
        $ ./devVR/LenstraZ.out
*/

#include <math.h> // standard libraries
#include <fstream>
#include <iostream>
#include <sstream>
#include <iomanip>
// #include <windows.h>
#include <unistd.h>
#include <stdio.h>
#include <cstdio>
#include <stdlib.h>
#include <cstdlib>
#include <filesystem>
// #include <mmsystem.h>
#include <time.h>
#include <ctime>
#include <chrono>
#include <string>
#include <thread>
#include <array>
#include <sys/time.h>
#include <sys/resource.h>
#include <sys/stat.h>
#include <sys/types.h>

#include "NTL/ZZ.h" // NTL Libraries
```

```
#include "NTL/ZZ_p.h"
#include "NTL/ZZ_pX.h"
#include "NTL/ZZX.h"
#include "NTL/vec_ZZ.h"
#include <NTL/ZZ_pXFactoring.h>
#include <NTL/BasicThreadPool.h>
NTL_CLIENT

#include "PerfectPower.h" // Personal Headers
#include "Euler.h"
#include "Carmichael.h"
#include "CongruenceZ.h"

std::string getDate() {
    auto t = std::time(nullptr);
    auto tm = *std::localtime(&t);

    std::ostringstream oss;
    oss << std::put_time(&tm, "%Y-%m-%d");
    auto date = oss.str();

    return date;
}

std::string getTime() {
    auto t = std::time(nullptr);
    auto tm = *std::localtime(&t);

    std::ostringstream oss;
    oss << std::put_time(&tm, "%H-%M-%S");
    auto time = oss.str();

    return time;
}

std::string getDateTime() {
    auto t = std::time(nullptr);
    auto tm = *std::localtime(&t);

    std::ostringstream oss;
    oss << std::put_time(&tm, "%Y-%m-%d-%H-%M-%S");
    auto datetime = oss.str();

    return datetime;
}

std::string getFilename() {
    std::string fldr = "logs/Z/";
    std::string prfx = "LnstrZ-";
    std::string sffx = getDateTime();
    std::string extn = ".csv";

    std::string filename = fldr + prfx + sffx + extn;

    return filename;
}

std::string filename = getFilename();
std::ofstream perflog(filename, std::ios::app); // output result
   into file

inline void fileWrite(const ZZ& n, const unsigned int& cores, const
   bool& PRIME, const long& time, const std::string& other) {
    perflog << n << "," << cores << "," << PRIME  << "," << time <<
       "," << other << "\n";
}

inline bool Lenstra (const ZZ& n) {
    if(n < 1){
        std::printf("Integer n needs to be positive.\n");
        return false;
    }
    else if(n == 1){
        std::printf("1 is neither prime or composite.\n");
        return false;
    }
    else if(n == 2){
        std::printf("2 is prime.\n");
        return true;
    }
    else if(n == 3){
        std::printf("3 is prime.\n");
        return true;
    }

    std::cout << "n = " << n << "\n\n";

    // start timing
    auto start = std::chrono::high_resolution_clock::now();

    // Test if n is a perfect power
    int PP = PerfectPower(n);

    // returns 1 if n is a perfect power, 0 otherwise;
    if(PP == 1){
        auto finish = std::chrono::high_resolution_clock::now();
        auto duration = finish - start;
        auto time = std::chrono::duration_cast<std::chrono::
           milliseconds>(duration).count();

        std::printf("%ld is not prime.\n",to_long(n));
        std::printf("%ld is a perfect power.\n",to_long(n));
        std::printf("Time taken: %ld milliseconds\n",time);

        std::string note = std::to_string(to_long(n)) + " is a
           perfect power";
        fileWrite(n,ncores,false,time,note);

        return false;
    }

    // Find a suitable r
    ZZ r = to_ZZ(2);
    ZZ R;
    ZZ r1;

    while(r < n){
        ZZ R = GCD(r, n);
        if(R != 1 ){
            auto finish = std::chrono::high_resolution_clock::now();
            auto duration = finish - start;
            auto time = std::chrono::duration_cast<std::chrono::
               milliseconds>(duration).count();

            std::printf("%ld is not prime.\n",to_long(n));
            std::printf("%ld is a divisor.\n",to_long(R));
            std::printf("Time taken: %ld milliseconds\n",time);

            std::string note = std::to_string(to_long(R)) + " is a
               divisor";
            fileWrite(n,ncores,false,time,note);

            return false;
            break;
        }
        else {
            ZZ v = to_ZZ(floor(power_long(to_long(log(n)), 2)));

            // order of n mod r is bigger than v;
            int p = 0;
            ZZ_p::init(r); // calculate mod r

            while(v <= r){
                ZZ x = to_ZZ(power_long(to_long(n), to_long(v))); //
                    calculates x = n^v
                ZZ_p z = to_ZZ_p(x);
                if(z == to_ZZ_p(1)){
                    r1 = r; // store value of r
                    r = n + 1;
                    break;
                }
                else{
                    v = v + 1;
                }
            }
        }
        r = r + 1;
    }

    r = r1;
    std::printf("r = %ld\n",to_long(r));

    ZZ r2 = Euler(to_long(r));
    std::printf("Phi(%ld) = %ld\n",to_long(r),to_long(r2));
    // ZZ r2 = Carmichael(to_long(r));
    // std::printf("Lambda(%ld) = %ld\n",to_long(r),to_long(r2));

    long a = to_long(r2 - 1);
    long f = CongruenceZ(n,r,r2,a);

    if(f == 0){
        auto finish = std::chrono::high_resolution_clock::now();
        auto duration = finish - start;
        auto time = std::chrono::duration_cast<std::chrono::
           milliseconds>(duration).count();

        std::printf("%ld is prime.\n",to_long(n));
        std::printf("Time taken: %ld milliseconds\n",time);

        std::string note = "a = " + std::to_string(a) + "; End: a =
           " + std::to_string(f) + "; r = " + std::to_string(to_long
           (r)) + "; phi(r) = " + std::to_string(to_long(r2));
        // std::string note = "a = " + std::to_string(a) + "; End: a
            = " + std::to_string(f) + "; r = " + std::to_string(
           to_long(r)) + "; lambda(r) = " + std::to_string(to_long(
           r2));
        fileWrite(n,ncores,true,time,note);

        return true;
    }
    else {
        auto finish = std::chrono::high_resolution_clock::now();
        auto duration = finish - start;
        auto time = std::chrono::duration_cast<std::chrono::
           milliseconds>(duration).count();

        std::printf("%ld is not prime.\n",to_long(n));
        std::printf("The a which fails is %ld\n",f);
        std::printf("Time taken: %ld milliseconds\n",time);

        std::string note = "a = " + std::to_string(a) + "; End: a =
           " + std::to_string(f) + "; r = " + std::to_string(to_long
           (r)) + "; phi(r) = " + std::to_string(to_long(r2));
        // std::string note = "a = " + std::to_string(a) + "; End: a
            = " + std::to_string(f) + "; r = " + std::to_string(
            to_long(r)) + "; lambda(r) = " + std::to_string(to_long(
            r2));
        fileWrite(n,ncores,false,time,note);

        return false;
    }

}

int main (int argc, char * argv[]) {

    perflog << "Int, Cores, Prime (T/F), Time (milliseconds),
       Comments\n";

    ZZ p = conv<ZZ>("23456789");

    bool prime = Lenstra(p);

}
```

C.3.2 CongruenceZ.h

Listing C.4: The congruence testing header associated with the Z version of AKS. This function is essentially a clone of `CongruenceZnx` in Listing C.2 but calls the `ZZpPowMod` function from `biSegMultiplyZZpX.h` in Listing C.6 which uses the binary segmentation algorithms for polynomial multiplication.

```

#include "NTL/ZZ_p.h"
#include "NTL/ZZ_pX.h"
NTL_CLIENT

#include "biSegMultiplyZZpX.h"

unsigned int ncores = std::thread::hardware_concurrency(); //
   machine cores - may return 0 when not able to detect

long CongruenceZ(const ZZ& n, const ZZ& r, const ZZ& r2, const long&
    a) {
    // congruence test of polynomials in large integer form

    ZZ_p::init(n);                          // initialise mod n

    printf("\nCalculating x^r - 1 (mod n) ...\n");
    ZZ_pX b = ZZ_pX(to_long(r), 1) - 1; // b = x^r - 1 (mod n);
    printf("Done.\n");

    printf("Initialising x (mod n) ...\n");
    ZZ_pX e = ZZ_pX(1, 1);                  // e = x (mod n)
    printf("Done.\n");

    printf("Calculating x^n (mod (x^r - 1), n) ...\n");
    ZZ_pX d = ZZpPowMod(e, n, b);         // d = x^n (mod b, n)
    printf("Done.\n");

    printf("\nCommencing congruence tests:\n\n");
    for(long j = 1; j <= a; ++j){
        printf("a = %ld\n",j);

        ZZ_pX c = ZZ_pX(1, 1) - j ;          // c = x - a (mod n);
        ZZ_pX f = ZZpPowMod(c, n, b);         // f = (x - a)^n (mod b, n) - LHS
        ZZ_pX g = d - j;                     // g = x^n - a (mod b, n) - RHS

        if(f != g){
            return(j); // n is not prime
        }
    }

    return(0); // n is prime
}
```

C.4 Miscellaneous Headers

C.4.1 biSegMultiplyZZX.h

Listing C.5: A set of functions developed from scratch for computing various operations on polynomials of type `NTL::ZZX`. This header is largely useless in the final program however it was a milestone in the development of the binary segmentation and *powermod* algorithms.

```

#include <math.h>
#include <vector>
#include <algorithm>

#include "NTL/ZZ.h"
#include "NTL/ZZ_p.h"
#include "NTL/ZZ_pX.h"
NTL_CLIENT

ZZ powMod(ZZ a, ZZ n, const ZZ& b) {
    // calculates a^n (mod b) in O(log n)

    ZZ ans = ZZ(1);          // Initialise answer

    while (n > 0) {
        if (n % 2 == 1) {    // if (n is odd) then
            ans = (ans * a) % b;
        }
        a = (a * a) % b;     // a = a^2 (mod b)
        n /= 2;              // n = n/2
    }

    return ans;
}

ZZ getMaxCoeff(const ZZX& f) {
    /*
        Returns the largest integer coefficient, f_i, of the input
        polynomial, f(x), in O(D) time where D is the number of
           terms
        in f(x), including those with coefficients equal to zero.
    */

    ZZ f_i = ConstTerm(f);

    for (long i = 1; i <= deg(f); ++i) {
        if (f_i < coeff(f,i)) {
            f_i = coeff(f,i);
        }
        else {
            continue;
        }
    }

    return f_i;
}

ZZ evaluate(const ZZX& f, const ZZ& x) {
    /*
        Returns the integer result when the polynomial f(y) (mod p)
        is evaluated for y = x in O(D) time where D is the number of
        terms in f(x) (mod p), including those with coefficients
        equal to zero.
    */

    long fDeg = deg(f);
    ZZ ans = ConstTerm(f);

    ZZ xpow = ZZ(1);
    for (long i = 1; i <= fDeg; ++i) {
        xpow = xpow*x;
        ans += coeff(f,i)*xpow;
    }

    return ans;
}

ZZX polyMultiply(const ZZX& f, const ZZX& g) {
    /*
        Returns the polynomial product s(x) = f(x) * g(x), using
        binary segmentation, in O(D) time where D is the number of
        terms in s(x), including those with coefficients equal to
           zero.
    */

    ZZ termsF = to_ZZ(deg(f)+1);
    ZZ termsG = to_ZZ(deg(g)+1);

    ZZ maxTermFG = to_ZZ(std::max(termsF,termsG));
    ZZ maxCoeffF = getMaxCoeff(f);
    ZZ maxCoeffG = getMaxCoeff(g);

    ZZ rhs = maxTermFG * maxCoeffF * maxCoeffG;          // rhs = max
       (U,V) * max(f_j) * max(g_k)

    long b = 1;
    ZZ lhs = (power2_ZZ(b)) - 1;                         // lhs = 2^b
       -1
    while (lhs <= rhs) {                                 // Choose b
       such that 2^b     1 > max(U,V) * max(f_j) * max(g_k)
        b = b + 1;
        lhs = (power2_ZZ(b)) - 1;
    }

    ZZ X = lhs;
    ZZ F = evaluate(f,X);                                // evaluate
       f(x) for x = 2^b - 1
    ZZ G = evaluate(g,X);                                // evaluate
       g(x) for x = 2^b - 1

    ZZ m = F * G;                                        // Integer
       multiply

    // long fgDeg = deg(f) + deg(g);                        // Degree
        of polynomial product
    long fgTrm = deg(f) + deg(g) + 1;                    // Terms in
       polynomial product
    // long fgTrm = termsF + termsG - 1;                        //
       Terms in polynomial product
    ZZ s[fgTrm];                                         // Store
       coefficients and constant in an array

    s[0] = m % lhs;                                      //
       Reassemble coefficients into signal
    if (s[0] > lhs/2) {                                  // Extract
       next b bits: s_i = floor( m/(2 b 1 )^i ) mod 2^b     1
        s[0] = s[0] - lhs;
        m = m + lhs;
    }
    for (long i = 1; i < fgTrm; ++i) {
        m = m / lhs;                                     // N.B. --
           NTL "/" operator floors result by default
        s[i] = m % lhs;

        if (s[i] > lhs/2) {
            s[i] = s[i] - lhs;
            m = m + lhs;
        }
    }

    ZZX polyProduct;                                        // Base-b digits of m are desired coefficients
    polyProduct.SetLength(fgTrm);                           // set the length of the underlying coefficient vector to number of Terms in polynomial product

    for (long j = 0; j < fgTrm; ++j) {
        SetCoeff(polyProduct,j,s[j]);
    }

    polyProduct.normalize();                                // remove leading zeros on coefficients

    return polyProduct;
}

ZZX polyMulMod(const ZZX& f, const ZZX& g, const ZZX& h) {
    /*
        Returns the polynomial product s(x) = f(x) * g(x) (mod h(x)),
        using binary segmentation, in O(D) time where D is the number
        of terms in s(x), including those with coefficients equal to zero.
    */

    ZZ termsF = to_ZZ(deg(f)+1);
    ZZ termsG = to_ZZ(deg(g)+1);

    ZZ maxTermFG = to_ZZ(std::max(termsF,termsG));
    ZZ maxCoeffF = getMaxCoeff(f);
    ZZ maxCoeffG = getMaxCoeff(g);

    ZZ rhs = maxTermFG * maxCoeffF * maxCoeffG;             // rhs = max(U,V) * max(f_j) * max(g_k)

    long b = 1;
    ZZ lhs = (power2_ZZ(b)) - 1;                            // lhs = 2^b -1
    while (lhs <= rhs) {                                     // Choose b such that 2^b   1 > max(U,V) * max(x_i) * max(y_k)
        b = b + 1;
        lhs = (power2_ZZ(b)) - 1;
    }

    ZZ X = lhs;
    ZZ F = evaluate(f,X);                              // evaluate f(x) for x = 2^b - 1
    ZZ G = evaluate(g,X);                              // evaluate g(x) for x = 2^b - 1

    ZZ m = F * G;                                      // Integer multiply

    // long fgDeg = deg(f) + deg(g);                      // Degree of polynomial product
    long fgTrm = deg(f) + deg(g) + 1;                  // Terms in polynomial product
    ZZ s[fgTrm];                                       // Store coefficients and constant in an array

    s[0] = m % lhs;                                    // Reassemble coefficients into signal
    if (s[0] > lhs/2) {                                // Extract next b bits: s_i = floor( m/(2 b 1 )^i ) mod 2^b     1
        s[0] = s[0] - lhs;
        m = m + lhs;
    }
    for (long i = 1; i < fgTrm; ++i) {
        m = m / lhs;                                   // N.B. -- NTL "/" operator floors result by default
        s[i] = m % lhs;

        if (s[i] > lhs/2) {
            s[i] = s[i] - lhs;
            m = m + lhs;
        }
    }

    ZZX polyProduct;                                   // Base-b digits of m are desired coefficients
    polyProduct.SetLength(fgTrm);                      // set the length of the underlying coefficient vector to number of Terms in polynomial product

    for (long j = 0; j < fgTrm; ++j) {
        SetCoeff(polyProduct,j,s[j]);
    }

    polyProduct.normalize();                           // remove leading zeros on coefficients

    ZZX polyProdMod = polyProduct % h;

    return polyProdMod;
}
```

```

ZZX polyMulModN(const ZZX& f, const ZZX& g, const ZZ& n) {
    /*
        Returns the polynomial product s(x) = f(x) * g(x) (mod n), using
        binary segmentation, in O(D) time where D is the number of terms
        in s(x), including those with coefficients equal to zero.
    */

    ZZ termsF = to_ZZ(deg(f)+1);
    ZZ termsG = to_ZZ(deg(g)+1);

    ZZ maxTermFG = to_ZZ(std::max(termsF,termsG));
    ZZ maxCoeffF = getMaxCoeff(f);
    ZZ maxCoeffG = getMaxCoeff(g);

    ZZ rhs = maxTermFG * maxCoeffF * maxCoeffG;     // rhs = max(U,V) * max(f_j) * max(g_k)
    long b = 1;
    ZZ lhs = (power2_ZZ(b)) - 1;                    // lhs = 2^b-1
    while (lhs <= rhs) {                            // Choose b such that 2^b      1 > max(U,V) * max(x_i) * max(y_k)
        b = b + 1;
        lhs = (power2_ZZ(b)) - 1;
    }

    ZZ X = lhs;
    ZZ F = evaluate(f,X);                           // evaluate f(x) for x = 2^b - 1
    ZZ G = evaluate(g,X);                           // evaluate g(x) for x = 2^b - 1

    ZZ m = F * G;                                   // Integer multiply

    // long fgDeg = deg(f) + deg(g);                   // Degree of polynomial product
    long fgTrm = deg(f) + deg(g) + 1;               // Terms in polynomial product
    ZZ s[fgTrm];                                    // Store coefficients and constant in an array

    s[0] = m % lhs;                                 // Reassemble coefficients into signal
    if (s[0] > lhs/2) {                             // Extract next b bits: s_i = floor( m/(2 b 1 )^i ) mod 2^b      1
        s[0] = s[0] - lhs;
        m = m + lhs;
    }
    for (long i = 1; i < fgTrm; ++i) {
        m = m / lhs;                          // N.B. -- NTL "/" operator floors result by default
        s[i] = m % lhs;

        if (s[i] > lhs/2) {
            s[i] = s[i] - lhs;
            m = m + lhs;
        }
    }

    ZZX polyProdModN;                         // Base-b digits of m are desired coefficients
    polyProdModN.SetLength(fgTrm);            // set the length of the underlying coefficient vector to number of Terms in polynomial product

    for (long j = 0; j < fgTrm; ++j) {
        SetCoeff(polyProdModN,j,(s[j] % n));  // s[j] (mod n)
    }

    polyProdModN.normalize();                 // remove leading zeros on coefficients

    return polyProdModN;
}

ZZX polyPowMod(ZZX a, ZZ n, ZZX b) {
    /*
        Calculates a(x)^n (mod b(x)) in O((log n)*O(D)) time.
    */

    ZZX ans;                    // Initialise answer
    SetCoeff(ans,0,1);

    while (n > 0) {
        if (n % 2 == 1) {       // if (n is odd) then
            ans = polyMultiply(ans,a) % b;
        }
        a = polyMultiply(a,a);  // a = a^2 (mod b)
        n /= 2;                 // n = n/2
    }

    return ans;
}
```

C.4.2 biSegMultiplyZZpX.h

Listing C.6: A set of functions developed from scratch for computing various operations on polynomials of type `NTL::ZZ_pX`. This header is contains the final versions of the *powermod* and binary segmentation algorithms called from `CongruenceZ.h` in Listing C.4.

```
#include <math.h>
#include <vector>
#include <algorithm>

#include "NTL/ZZ.h"
#include "NTL/ZZ_p.h"
#include "NTL/ZZ_pX.h"
NTL_CLIENT

ZZ_p getMaxCoeff(const ZZ_pX& f) {
    /*
        Returns the largest integer coefficient, f_i, of the input
        polynomial, f(x) (mod p), in O(D) time where D is the number
        of terms in f(x) (mod p), including those with coefficients
        equal to zero.
    */

    ZZ_p f_i = ConstTerm(f);
    ZZ comp = rep(ConstTerm(f));

    for (long i = 1; i <= deg(f); ++i) {
        if (comp < rep(coeff(f,i))) {
            f_i = coeff(f,i);
            comp = rep(coeff(f,i));
        }
        else {
            continue;
        }
    }

    return f_i;
}

ZZ evaluate(const ZZ_pX& f, const ZZ& x) {
    /*
        Returns the integer result when the polynomial f(y) (mod p)
        is evaluated for y = x in O(D) time where D is the number of
        terms in f(x) (mod p), including those with coefficients
        equal to zero.
    */

    long fDeg = deg(f);
    ZZ ans = rep(ConstTerm(f));

    ZZ xpow = ZZ(1);
    for (long i = 1; i <= fDeg; ++i) {
        xpow = xpow*x;
        ans += rep(coeff(f,i))*xpow;
    }

    return ans;
```

```
}

ZZ_pX ZZpXmultiply(const ZZ_pX& f, const ZZ_pX& g) {
    /*
        Returns the polynomial product s(x) =  f(x) * g(x) (mod p),
        using binary segmentation, in O(D) time where D is the number
        of terms in s(x), including those with coefficients equal to zero.
    */

    // ZZ termsF = to_ZZ(deg(f)+1);
    // ZZ termsG = to_ZZ(deg(g)+1);
    long termsF = deg(f)+1;
    long termsG = deg(g)+1;

    ZZ maxTermFG = to_ZZ(std::max(termsF,termsG));
    ZZ_p maxCoeffF = getMaxCoeff(f);
    ZZ_p maxCoeffG = getMaxCoeff(g);
    // ZZ maxCoeff = ZZ_p::modulus();

    ZZ rhs = maxTermFG * rep(maxCoeffF) * rep(maxCoeffG);   // rhs = max(U,V) * max(f_j) * max(g_k)
    // ZZ rhs = maxTermFG * sqr(maxCoeff);                      // rhs = max(U,V) * max(f_j) * max(g_k)

    long b = 1;
    ZZ lhs = (power2_ZZ(b)) - 1;                            // lhs = 2^b-1
    while (lhs <= rhs) {                                    // Choose b such that 2^b   1 > max(U,V) * max(f_j) * max(g_k)
        b = b + 1;
        lhs = (power2_ZZ(b)) - 1;
    }

    ZZ X = lhs;
    ZZ F = evaluate(f,X);                                   // evaluate f(x) for x = 2^b - 1
    ZZ G = evaluate(g,X);                                   // evaluate g(x) for x = 2^b - 1

    ZZ m = F * G;                                           // Integer multiply

    // long fgDeg = deg(f) + deg(g);                            // Degree of polynomial product
    // long fgTrm = deg(f) + deg(g) + 1;                        // Terms in polynomial product
    long fgTrm = termsF + termsG - 1;                       // Terms in polynomial product
    ZZ_p s[fgTrm];                                          // Store
```

```
        coefficients and constant in an array

    s[0] = to_ZZ_p(m % lhs);                              //
       Reassemble coefficients into signal
    if (rep(s[0]) > lhs/2) {                              //
       Extract next b bits: s_i = floor( m/(2 b 1 )^i ) mod 2^b
       1
        s[0] = s[0] - to_ZZ_p(lhs);
        m = m + lhs;
    }
    for (long i = 1; i < fgTrm; ++i) {
        m = m / lhs;                                      // N.B.
           -- NTL "/" operator floors result by default
        s[i] = to_ZZ_p(m % lhs);

        if (rep(s[i]) > lhs/2) {
            s[i] = s[i] - to_ZZ_p(lhs);
            m = m + lhs;
        }
    }

    ZZ_pX polyModProd;                                    // Base-
       b digits of m are desired coefficients
    polyModProd.SetLength(fgTrm);                         // set
       the length of the underlying coefficient vector to number of
       Terms in polynomial product

    for (long j = 0; j < fgTrm; ++j) {
        SetCoeff(polyModProd,j,s[j]);
    }

    polyModProd.normalize();                              //
       remove leading zeros on coefficients

    return polyModProd;
}

ZZ_pX ZZpPowMod(ZZ_pX a, ZZ n, const ZZ_pX& b) {
    /*
        Calculates a(x)^n (mod b(x), p) in O(log n) time.
    */

    ZZ_pX ans;                          // Initialise answer
    SetCoeff(ans,0,1);

    while (n > 0) {
        if (n % 2 == 1) {               // if (n is odd) then
            ans = ZZpXmultiply(ans,a);
            ans %= b;
        }
        a = ZZpXmultiply(a,a);          // a = a^2 (mod b)
        // a = sqr(a);                    // a = a^2 (mod b)
        a %= b;

        n /= 2;                        // n = n/2
    }

    return ans;
}
```

C.4.3 Carmichael.h

Listing C.7: A set of functions developed from scratch for computing the prime factors and LCM of any integer, as well as Carmichael's reduced totient, $\lambda(n)$. Current implementations are sub-optimal and some improvements could be made very easily (enable/uncomment the print statements and run to see how). Use of this header is optional but recommended for use in place of the `Euler` function from `Euler.h`. Development and use of this function was inspired by Egan (2005).

```

#include <vector>
#include <algorithm>

#include "NTL/ZZ.h"
#include "NTL/ZZ_p.h"
NTL_CLIENT

bool isPrime(long n) {

    long i = 2;
    while (i <= n/2) {

        if (n % i == 0) {
            return false;
        }

        i += 1;
        i = NextPrime(i);
    }

    return true;
}

void primeFactors(long n, vector<long>& p, vector<long>& e) {

    // std::cout << "\nChecking if " << n << " is prime...\n";

    if (isPrime(n)) {
        p.push_back(n);
        e.push_back(1);
        // std::cout << n << " is PRIME.\n";
        return;
    }

    // std::cout << n << " is NOT PRIME.\nCalculating prime factors
```

```
        and exponents...\n\n";

    long idx = 0;
    long i = 2, end = n/2;
    while (i <= end) {
        // std::cout << "i = " << i << "\n";
        // std::cout << "n = " << n << "\n";

        p.push_back(i);
        e.push_back(0);

        while (n % i == 0) {
            n /= i;
            e[idx] += 1;
        }

        idx +=1;

        i += 1;
        i = NextPrime(i);
    }

}

unsigned long long LCM(vector<long>& n) {

    unsigned long long lcm = 1;
    std::vector<long> p;
    std::vector<long> e;

    std::vector<long> b;
    std::vector<long> t;

    b.resize(n.size());
    t.resize(n.size());

    for (long i = 0; i < n.size(); ++i) {
        primeFactors(n[i],p,e);

        // std::cout << "Prime factors:\t";
        // for (int i = 0; i < p.size(); i++)
        //     std::cout << p[i] << "\t";

        // std::cout << "\nExponents:\t";
        // for (int i = 0; i < p.size(); i++)
        //     std::cout << e[i] << "\t";
        // std::cout << "\n";

        for (long j = 0; j < p.size(); ++j) {
            if (t[j] < e[j]) {
                t[j] = e[j];
            }
            if (b[j] < p[j]) {
                b[j] = p[j];
            }
        }

        p.clear();
        e.clear();
    }

    // std::cout << "\nLCM Prime factors:\t";
    // for (int i = 0; i < b.size(); i++)
    //     std::cout << b[i] << "\t";

    // std::cout << "\nLCM Exponents:\t\t";
    // for (int i = 0; i < t.size(); i++)
    //     std::cout << t[i] << "\t";
    // std::cout << "\n";

    for (long k = 0; k < t.size(); ++k) {
        lcm *= power_long(b[k],t[k]);
    }

    return lcm;
}

unsigned long long Phi(unsigned long long r){
    unsigned long long eu = 1;

    for (unsigned long long p = 2; p * p <= r; p += 2) {
        if (r % p == 0) {
            eu *= p - 1;
            r /= p;

            while (r % p == 0) {
                eu *= p;
                r /= p;
            }
        }

        if(p == 2) {
            --p;
        }
    }

    unsigned long long eu1 = eu;

    // now r is prime or 1
    if (r == 1) {
        return eu1;
    }
    else {
        return eu1 * (r - 1) ;
    }
}

long subLambda(const long& a, const long& b) {

    unsigned long long lambda;

    if (b <= 2 || a >= 3) {
        lambda = Phi(power_long(a,b));
        // std::cout << "\nsubLambda = phi(a^b) = " << a << "^" << b
             << " = " << lambda << "\n";
    }
    else if (b >= 3 && a == 2) {
        lambda = Phi(power_long(a,b))/2;
        // std::cout << "\nFinal: lambda = phi(a^b)/2 = " << a <<
           "^" << b << "/2 = " << lambda << "\n";
    }
    else {
        std::printf("\nError: Out of condition bounds.\n");
    }

    return lambda;
}

ZZ Carmichael(const long& n) {

    if (isPrime(n)) {
        return ZZ(Phi(n));
    }

    unsigned long long lambda;

    std::vector<long> p;
    std::vector<long> e;

    primeFactors(n,p,e);

    std::vector<long> comp;

    for (long i = 0; i < p.size(); ++i) {
        comp.push_back(subLambda(p[i],e[i]));
    }

    lambda = LCM(comp);

    return ZZ(lambda);
}
```

C.4.4 Euler.h

Listing C.8: A function for computing the Euler totient of n, $\phi(n)$. Reused from Li (2007) with minor modifications for performance optimisation. Use of this function is mandatory in the pre-computation stage of AKS but it is recommended that the lambda function in `Carmichael.h` be used in its place.

```

// Euler's phi function

ZZ Euler(long r){
    long eu = 1;

    for (long p = 2; p * p <= r; p += 2) {
        if (r % p == 0) {
            eu *= p - 1;
            r /= p;

            while (r % p == 0) {
                eu *= p;
                r /= p;
            }
        }

        if(p == 2) {
            --p;
        }
    }

    ZZ eu1 = to_ZZ(eu);

    // now r is prime or 1
    if (r == 1) {
        return eu1;
    }
    else {
        return eu1 * (r - 1) ;
    }
}
```

C.4.5 PerfectPower.h

Listing C.9: A function for deciding whether or not an integer input is a perfect power. Reused from Li (2007) with minor modifications for performance optimisation. Called from both main program files in the pre-computation stage of AKS.

```

// function to calculate if n = a^b
// takes input ZZ n and returns 1 if n is a perfect power, 0
   otherwise

#include <NTL/RR.h>

int PerfectPower(const ZZ& n){
    long b = 2;
    RR k = to_RR(log(n) / log(2));
    ZZ a;

    while(b <= to_long(k)){
        // b cannot be bigger than k
        long c = to_long(ceil((log(n) / log(2)) / to_long(b)));
        a = pow(2, c); // assign guess value for a

        while(power(a, b) > n){
            double d = to_double(((b - 1) * a + n / power(a, (b - 1)
               )) / b); // Apply Integer Newton's Method
            ZZ e = to_ZZ(floor(d));
            a = to_long(e); // adjust a
        }

        if(n == power(a, b)){
            // if n is a perfect power
            printf("n = a^b\n");
            printf("b = %ld\n",b);
            printf("a = %ld\n\n",to_long(a));
            return(1);
        }
        else{
            b = b + 1;
        }
    }

    if(n != power(a,b)){
        return(0); // n is not a perfect power
    }

    return(0);
}
```

C.5 Parallel Functions

C.5.1 CongruenceZnxHPC.h

Listing C.10: A basic implementation of congruence testing in parallel, used to assess parallel implementation strategies. The NTL thread pools module offers an excellent introduction to parallel computing and handles everything to to with memory allocation & partitioning of the domain of a using its own macros. In its current state, this header can be used on any machine with NTL-11.5.1, or later, installed and will automatically detect the number of cores on the machine using the command in line 10. This can be overridden by changing the `ncores` variable in line 14 up to the maximum number of cores available. If it is unable to detect the number of cores, the default is 1 so the program will run in serial on a single core. Use on laptops is possible, but not recommended for battery reasons.

```
1
2  #include <vector>
3  #include <numeric>
4
5  #include "NTL/ZZ_p.h"
6  #include "NTL/ZZ_pX.h"
7  #include <NTL/BasicThreadPool.h>
8  NTL_CLIENT
```

```

unsigned int ncores = std::thread::hardware_concurrency(); // machine cores - may return 0 when not able to detect

long CongruenceZnxHPC(const ZZ& n, const ZZ& r, const ZZ& r2, const long& a){
    // congruence test of polynomials in regular form
    SetNumThreads(ncores); // number of threads - should correspond to the number of available cores on your machine
    std::cout << AvailableThreads() << " of " << ncores << " threads available.\n";

    ZZ_p::init(n);                          // initialise mod n
    ZZ_pContext context;
    context.save();

    printf("\nCalculating x^r - 1 (mod n) ...\n");
    ZZ_pX b = ZZ_pX(to_long(r), 1) - 1; // b = x^r - 1 (mod n);
    printf("Done.\n");

    printf("Initialising x (mod n) ...\n");
    ZZ_pX e = ZZ_pX(1, 1);                  // e = x (mod n)
    printf("Done.\n");

    printf("Calculating x^n (mod (x^r - 1), n) ...\n");
    ZZ_pX d = PowerMod(e, n, b);            // d = x^n (mod b, n)
    printf("Done.\n");

    vector <long> test;
    test.reserve(ncores);

    printf("\nCommencing congruence tests:\n\n");
    long first, last;
    NTL_EXEC_RANGE(a,first,last)

        context.restore();

        for(long j = first; j < last; ++j){
            if (j < last) {                                // ensures each thread does not continue past j = a
                printf("a = %ld\n",(j + 1));

                ZZ_pX c = ZZ_pX(1, 1) - (j + 1);     // c = x - a (mod n);
                ZZ_pX f = PowerMod(c, n, b);         // f = (x - a)^n (mod b, n) - LHS
                ZZ_pX g = d - (j + 1);               // g = x^n - a (mod b, n) - RHS

                if(f != g){
                    test.push_back((j + 1)); // n is not prime
```

```
                        break;
                    }
                }
            }

    NTL_EXEC_RANGE_END

    if (test.size() == 0) {
        return 0; // n is prime
    }
    else {
        long smallest = test.at(0);
        for (int k = 0; k < test.size(); ++k) {
            if (test.at(k) < smallest) {
                smallest = test.at(k);
            }
        }

        return smallest;
    }
}
```

Appendix D

Ethics Assessment

D.1 Comments on Relevant Ethical Issues

The author believes that there currently exist little to no ethical issues associated with this research. No experiments involving participants are necessary therefore none will be recruited for this work. The ethical implications of any findings and outcomes of this work will be raised and documented should they come to light. The author acknowledges that there are environmental costs related to the energy requirements involved with high performance computing and every effort will be made to limit waste and mitigate this issue. Ethics of this research and its methods will be continuously reviewed and, should any arise, addressed in the dissertation.

D.2 Department of Computer Science 12 Point Ethics Checklist

The pages which follow are reserved for a completed version of the University of Bath Department of Computer Science 12 Point Ethics Checklist.

UNIVERSITY OF BATH

Department of Computer Science

12-Point Ethics Checklist for UG and MSc Projects

Student	Vincent Jack Renders (vr317)
Academic Year or Project Title	Implementing the AKS primality test in CUDA (2020-2021)
Supervisor	Professor James H. Davenport (masjhd)

Does your project involve people for the collection of data other than you and your supervisor(s)? ~~YES~~ / [✓ NO]

If the answer to the previous question is YES, you need to answer the following questions, otherwise you can ignore them.

This document describes the 12 issues that need to be considered carefully before students or staff involve other people ('participants' or 'volunteers') for the collection of information as part of their project or research. Replace the text beneath each question with a statement of how you address the issue in your project.

1. *Will you prepare a Participant Information Sheet for volunteers?* YES / NO

   This means telling someone enough in advance so that they can understand what is involved and why – it is what makes informed consent informed.

2. *Will the participants be informed that they could withdraw at any time?* YES / NO

   All participants have the right to withdraw at any time during the investigation, and to withdraw their data up to the point at which it is anonymised. They should be told this in the briefing script.

3. *Will there be any intentional deception of the participants?* YES / NO

   Withholding information or misleading participants is unacceptable if participants are likely to object or show unease when debriefed.

4. *Will participants be de-briefed?* YES / NO

   The investigator must provide the participants with sufficient information in the debriefing to enable them to understand the nature of the investigation. This phase might wait until after the study is completed where this is necessary to protect the integrity of the study.

5. *Will participants voluntarily give informed consent?* YES / NO

Participants MUST consent before taking part in the study, informed by the briefing sheet. Participants should give their consent explicitly and in a form that is persistent –e.g. signing a form or sending an email. Signed consent forms should be kept by the supervisor after the study is complete. If your data collection is entirely anonymous and does not include collection of personal data you do not need to collect a signature. Instead, you should include a checkbox, which must be checked by the participant to indicate that informed consent has been given.

6. *Will the participants be exposed to any risks greater than those encountered in their normal work life (e.g., through the use of non-standard equipment)?* YES / NO

Investigators have a responsibility to protect participants from physical and mental harm during the investigation. The risk of harm must be no greater than in ordinary life.

7. *Will you be offering any incentive to the participants?* YES / NO

The payment of participants must not be used to induce them to risk harm beyond that which they risk without payment in their normal lifestyle.

8. *Will you be in a position of authority or influence over any of your participants?* YES / NO

A position of authority or influence over any participant must not be allowed to pressurise participants to take part in, or remain in, any experiment.

9. *Will any of your participants be under the age of 16?* YES / NO

Parental consent is required for participants under the age of 16.

10. *Will any of your participants have an impairment that will limit Their understanding or communication?* YES / NO

Additional consent is required for participants with impairments.

11. *Will the participants be informed of your contact details?* YES / NO

All participants must be able to contact the investigator after the investigation. They should be given the details of the Supervisor as part of the debriefing.

12. Will you have a data management plan for all recorded data? YES / NO

Personal data is anything which could be used to identify a person, or which can be related to an identifiable person. All personal data (hard copy and/or soft copy) should be anonymized (with the exception of consent forms) and stored securely on university servers (not the cloud).