Benchmarking of Integer Factorisation Algorithms on Semiprimes

Student Name: T. Zavileiskii
Supervisor Name: P. Davies
Submitted as part of the degree of BSc Natural Sciences to the
Board of Examiners in the Department of Computer Sciences, Durham University

Abstract—The paper considers several integer factorisation algorithms: Trial Division, Pollard's Rho, Elliptic Curve Method, Pollard's P-1, and Quadratic Sieve, and explores the former 3 in detail. Several implementations of 3 algorithms are presented and benchmarked on semiprimes, integers with 2 prime factors. The implementations cover both theoretical and practical details. The implementations go from basic versions, programmed in Python, to C++ where they are executed on single and multiple threads, and ending with implementations on a GPU with CUDA. Out of the attempted algorithms, the best performing implementation was ECM with 2 stages and Montgomery curves. The best overall implementation was the PARI mathematical library, which used an ensemble of algorithms and outperformed all implemented algorithms and other considered mathematical software.

Index Terms—Algebraic algorithms, Benchmarking, Mathematical Software, Parallel and vector implementations

1 Introduction

1.1 Problem Statement and Motivation

Integer Factorisation is a problem, where input is a composite integer n and output is another integer $1 < q < n$ such that $q|n$. There is currently no known algorithms which run in polynomial time, and the problem is in NP as solution can be trivially checked: given q and $p = n/q$ verify $p * q = n$.

The importance of factorisation rose with the adoption of the RSA encryption [1], as an efficient factorisation algorithm would break it (Appendix A), and make a lot of encrypted data vulnerable for the attackers. Already, old key sizes of 512 bits are within reach of the modern factorisation capabilities with 1024 bits being close, as the most recent record was factoring an RSA-250, 829 bit integer in 2020 [2]. It shows that exploration of factorisation limits is important to check which data is potentially vulnerable, to avoid using potentially insecure encryption algorithms, and provide lower bound for safe key sizes.

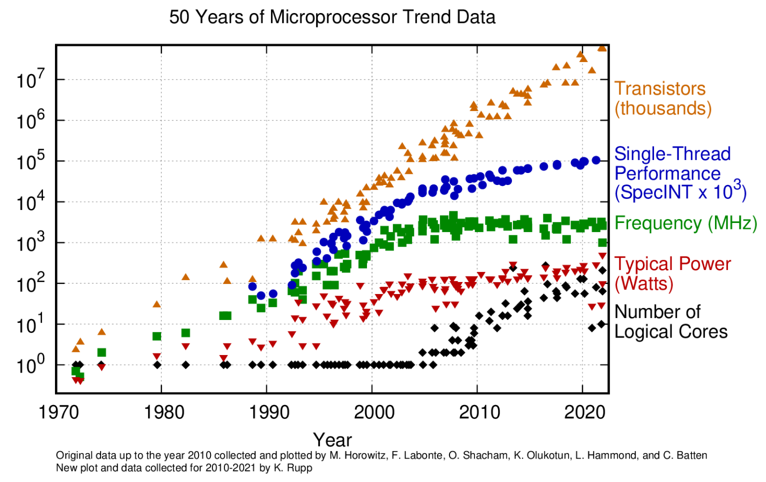


Fig. 1. Hardware Development Graph, [3]

1.2 Project Aims

The **research question** is, "what is the most efficient way to factorise semiprimes?" The paper will focus on both theoretical improvements and parallelisation to determine which algorithms can benefit from parallelism the most. It is important in the view of the recent trend in the hardware to focus on the parallel execution to maximise processing bandwidth rather than improving single thread performance or clock speeds. The trend is specifically prevalent in the 2010's with the large rise in use of GPUs for the general purpose computing [3]. As fig 1 shows, the clock frequency stopped improving since mid-2000s, with all the transistor increase being spent on the number of logical cores, making experimentation with adopting algorithms for parallel execution especially relevant.

The project deliverables consist of the benchmark performance data, and can be split into 3 categories:

- **Basic:** Textbook implementations of the algorithms with a high-level language like Python. This serves as a baseline to demonstrate how sensible the algorithm is to tuning and implementation details.
- **Intermediate:** Parallelised implementations of the algorithms written in a compiled language like C++.
- **Advanced:** Implementation of the optimised algorithms on a GPU with CUDA.

The project implements both exponential algorithms like Trial Division, Pollard's Rho, and Pollard's P-1, as well as a sub exponential Elliptic Curve Method (ECM) and

Quadratic Sieve. Relatively small pool of algorithms was selected to be able to provide enough attention to each of them as every algorithm typically has several possible optimisations, which can be tried.

Each algorithm will be implemented first in Python in its simplest text book form, then implemented in C++ and parallelised, with variations tested at this stage to determine the most promising one to be implemented for a GPU. In the end, algorithm is implemented on a GPU with CUDA.

The paper first contains related work with theoretical discussion of the factorisation algorithms, their variations, as well as providing a few other implementations. Then iterative process of algorithm improvement is presented, discussing each steps for each algorithm as well as the benchmarking method. The results section provides graphs with the . Evaluation discusses how obtained results follow the theoretical analysis and which optimisations had the largest impact on the performance. At last, the conclusion discusses the novel element in the paper and presents further work to be done in benchmarking the algorithms.

The paper also includes additional appendices, which are referred to, throughout the paper, but do not include essential content:

- **Appendix A:** RSA encryption scheme and how it can be broken with integer factorisation
- **Appendix B:** Contains Algorithm for generating Prime Table
- **Appendix C:** Includes command line arguments for running the benchmark

2 Related Work

2.1 Factorisation Algorithms

A great review of the main factorisation algorithms, their potential optimisations, along other relevant number theoretic algorithms can be found in the book Prime Numbers: A Computational Perspective by Carl Pomerance and Richard Crandall [4]. The book is a great starting point to learn about computational number theory from a theoretical side.

2.1.1 *Trial Division*

Trial Division is the most basic method to factorise an integer. To factorise n, it repeatedly tries to check $n \equiv 0$ (mod i) for $i \in [2, \lfloor\sqrt{n}\rfloor]$ [4, p. 117].

Algorithm 1 Trial Division

```
d = 2
for $i \in [2, \lfloor\sqrt{n}\rfloor]$ do
    rem = n (mod i)
    if rem $\neq 0$ then
        return i
    end if
end for
```

It works well for small factors as they are tested first, but becomes unfeasible due to exponential complexity $O(p)$. The algorithm is often used in combination with other algorithms to check for small factors.

Richard Brent in his paper from 1990 analysed methods for algorithm parallelisation, stating from the trail division [5]. If m trials (mod computations) are done in parallel on m machines, then the final time will be divided by m.

2.1.2 *Pollard's Rho*

Pollard's Rho is a Monte Carlo algorithm developed in the by John Pollard in 1975 [6]. It initialises u ← random_seed, and repeatedly applies a hash polynomial: $f(x) = x^2 + a$ (mod n) (a is usually 1) to u, so $u_{i+1} = f(u_i)$. At every step $\gcd(u_i - u_{2i}, n)$ is computed. If $1 < \gcd < n$ then a non-trivial factor is found. If a cycle is detected using Floyd's Cycle Finding Algorithm (i.e. $u_i = u_{2i}$) then u is randomly selected again.

Algorithm 2 Pollard's Rho

```
$f(x) = x^2 + 1$
$u, v \leftarrow$ random seed
while true do
    $u \leftarrow f(u)$
    $v \leftarrow f(f(v))$
    if $n > gcd(u - v, n) > 1$ then
        Return $\gcd(u - v, n)$
    else if $u \equiv v$ mod n then
        $u, v \leftarrow$ random seed
    end if
end while
```

The heuristic complexity of the algorithm was estimated as $O(p^{1/2}\log(N)^2)$, which is an improvement over trial division, but still exponential.

Richard Brent analysed several factorisation algorithm, including Pollard's Rho, providing a heuristic argument that parallelising the algorithm on m machines by choosing random u_0 and a on each machine will result in speed up by only $\sqrt{m}$ [5]. The sublinear improvement from parallel execution was shown in practice many times like in the 2013 paper, which ran the algorithm on multiple cores, parallelising it with OpenMP. The results showed the speed up by a factor of 2 between 1 and 2 threads, but only by a factor of 1.5 for between 2 and 4 threads, clearly signifying diminishing returns from increase in the number of parallel threads [7].

A potential variation of the algorithm was proposed by Crandall [8] to improve the speed up from running algorithm on m machines from $\sqrt{m}$ to $\log m^2/m$. He suggested making the machines to work not independently, but rather construct a large polynomial and use FFT polynomial evaluation scheme, previously proposed by Montgomery [9, p. 31], to construct the product. However, the proposed method was suggested to be faster only in the case with 1000s of parallel machines.

2.1.3 *Pollard's p-1*

Pollard's p-1 algorithm was developed by John Pollard in 1974 [10]. It relies on the Fermat's little theorem ($n^{p-1} = 1$ (mod p)) inside the ring $\mathbb{Z}/n$, estimating the heuristic complexity as $O(n^{1/2+\epsilon})$. The algorithm uses table of prime numbers and repeatedly raises the stating number, usually 2, to a power $p_i^{e_i} \leq B_1$ for maximal e_i. The procedure is repeated until the product of powers to which a is raised, M, is s.t. $n-1|M$, so by the Fermat's theorem, $2^M = 2^{d(p-1)} \equiv 1$ (mod n) where d is some constant, so potentially $\gcd(2^M - 1, 1) > 1$.

Algorithm may fail if all primes below B_1 were tried and factor was not found. It will happen in case there is $q|p-1$

Algorithm 3 Pollard's Rho

```
$B \leftarrow n^e$
Find primes $P = \{p_i \in \mathbb{Z} | p_i \text{ prime}, p_i \leq B\}$
$a \leftarrow 2$
for all $p_i \in P$ do
  $e_i \leftarrow \lfloor \log_{p_i}(B) \rfloor$
  $a \leftarrow a^{p_i^{e_i}}$
  if $\gcd(a-1, n) > 1$ then
    return $\gcd(a-1, n)$
  end if
end for
```

such that $q > B_1$. Either algorithm can be tried with higher B_1 or the 2nd stage can be used. There are several 2nd stages and one was proposed in the original paper. It finds B_2 and accelerates the cases where $p-1 = q\prod p_i^{a_i}$ and q is a prime such that $B\_2 > q > B\_1$. So the $p-1$ is product of many small primes $p < B_1$ and one large prime $B\_2 > q > B\_1$.

To execute the stage 2, for number $k = 2^M$, it finds k^{Q_i}, where Q_i is difference between primes q_i and q_{i-1}. Then points $q_i \in [B_2, B_1)$ can be checked with 1 multiplication as $2^{q_i M} = k^{q_{i-1}} * k^{Q_i}$.

2.1.4 *Elliptic Curve Method*

Elliptic Curve Method algorithm was developed by Lenstra in 1987, and it is an improvement over Pollard's p-1 algorithm. Instead of taking powers in a ring $\mathbb{Z}/n$, it adds points inside the Abelian group of points on an elliptic curves over the field $\mathbb{F}_\Bbbk$ [11]. Elliptic curve is defined with the equation $y^2 = x^3 + ax + b$ (Weierstrass Curve), for some constants a,b such that $4a^3 + 27b^2 \neq 0$ and $gcd(4a^3 + 27b^2, n) == 1$ (if latter is false, it will signify a factorisation). The group elements are points (x, y) that satisfy that equation. The group operation for points $(x_1, y_1) * (x_2, y_2) = (x_3, y_3)$ is defined as drawing a line l through (x_1, y_1) and (x_2, y_2), finding third point $(x_3, -y_3)$ where l intersects the curve and reflecting it along y axis to get (x_3, y_3) (fig 2).

The operation can be done with following equations:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1} \quad \mu = y_1 - \lambda x_1$$
$$x_3 = \lambda^2 - x_1 - x_2 \quad y_3 = -(\lambda * x_3 + \mu)$$

The equations also apply for finite fields like $\mathbb{F}_\Bbbk$. The scalar multiplication can be defined with repeated addition of the point to itself.

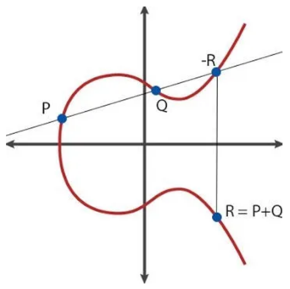


Fig. 2. Shows addition of 2 points P + Q = R [12]

The algorithm closely follows Pollard's p-1. Initially a random point on the curve is selected and it is multiplied by a scalar which is a prime raised to a power such that $p_i^{a_i} < B_1$. The algorithm uses the inverse step during point addition to find a factor, as it fails in cases $gcd(x_2 - x_1, n) \neq 1$ because $x_2 - x_1$ would not have a modular inverse mod n, but it signifies gcd is either a non-trivial factor of m or m itself. In the first case the algorithm is successful and in the other case, a new random curve is tried. The algorithm is successful when the order of a point divides M, like in the Pollard's p-1.

The algorithm has heuristic complexity of $O(\exp((\sqrt{2} + o(1))\sqrt{\ln p \ln \ln p})$ for the smallest factor p. Algorithm is more efficient than p-1 due to points on algebraic curves more likely to have order without large prime factors. Sub-exponential complexity combined with the low memory requirements make the algorithm very efficient for factorising large numbers with smaller factors.

Another advantageous property of the algorithm is that it can be efficiently parallelised with t machines. Brent also gave heuristic reason that ECM can be parallelised on m machines by trying m curves in parallel giving speed up by a factor of m [5].

The optimal choice of parameters B_1 and B_2 was studied by Silvermann, giving estimates for initial choice of B1 and B2 bounds [13]. The analysis covered the cases: maximising the probability of success for fixed amount of work, chance success per unit of work in the cases when the smallest factor is known, as well as suggesting when to apply other algorithms like Quadratic Sieve. It uses an integral of the Dickman's rho function which equals to the probability of x for $\lim\inf x$ having largest prime factor below $\rho(\alpha) = x^{\frac{1}{\alpha}}$. The paper defines another function $\mu(\alpha, \beta)$ which is probability of the largest factor of x being below $x^{\frac{\beta}{\alpha}}$ and the second largest factor below $x^{\frac{1}{\alpha}}$ and is defined as $\mu(\alpha, \beta) = \frac{1}{\alpha - \beta}\int_{\alpha-1}^{\alpha-\beta} \rho(t)dt$. The function μ is used to estimate the probability of success of a curve for bounds B_1, B_2 if $\alpha = \frac{\log B_1}{\log p}$, $\beta = \frac{\log B_1}{\log B_2}$. For example, it is used to find the optimal B_1 and B_2 per unit of work to by maximising $\mu(\alpha, \beta)/T$ where T is time taken for 1 curve.

An improvement to the algorithm was proposed by Montgomery, which uses projective coordinates and drops y from the calculations [14]. It uses 2 separate formulas for addition and doubling. A small downside is that addition requires [a-b]P to compute [a+b]P, but the main advantage is that it removes modular inverse in the point addition. It is given by the formulas the following formulas, so addition requires 5 modular multiplications and doubling requires 6 modular multiplications, which is way more efficient than computing 1 inverse:

Addition

$$X_{a+b} = Z_{a-b}((X_a - Z_a)(X_b + Z_b) + (X_a + Z_a)(X_b - Z_b))^2$$
$$Z_{a+b} = X_{a-b}((X_a - Z_a)(X_b + Z_b) - (X_a + Z_a)(X_b - Z_b))^2$$

Doubling

$$X_{2a} = (X_a + Z_a)^2(X_a - Z_a)^2$$
$$4X_aZ_a = (X_a + Z_a)^2 - (X_a - Z_a)^2$$
$$Z_{2a} = 4X_aZ_a((X_a - Z_a)^2 + \frac{A+2}{4}(4X_aZ_a))$$

The factor is found by checking $\gcd(z, n) \neq 1$, which will persist throughout point due to multiplicativity of gcd mod n. Another benefit of the method is that the curves in the Montgomery form would always have order divisible by 4, which can be improved by special initialisation of the curve and starting point P, using formulas proposed was first introduced for finding factors of large Fermat Numbers [15]. It will make the order of P divisible by 16, reducing the chances of large prime factors in the order.

A more modern parameterisation would be Edwards Curves developed by Harold Edward [16], with efficient point arithmetic using Inverted Coordinates and developed by the Daniel Bernstein and Tanja Lange [17]. It proposes formulas for Twisted and Inverted Twisted curves. Operations can be done in 10 modular multiplications for addition and 7 for doubling, which is worse than for Montgomery Pasteurisation. The main advantage of the proposed arithmetic is that it does not require the difference of points for addition and that it allows for efficient tripling with 13 modular multiplications.

One of the implementations of the ECM on GPU was presented in paper ECM at Work [18]. It utilises Edwards curves and investigates accelerating scalar multiplication by performing addition/substraction chains with different amount of available storage to save points, called windowing. It benchmarked windowing and and no windowing techniques on GPU, CPU while comparing the results with FPGA from a separate paper [19], concluding that GPU without windowing was the best platform for factorisation.

2.1.5 Quadratic Sieve

Quadratic Sieve (QS) was developed by Carl Pomerance in 1984 [20] and it is based on the fact that $x^2 = y^2 \pmod{n} \;=>\; (x-y)(x+y) = 0 \pmod{n}$ will signify factorisation, if $x \neq y \pmod{n}$. The algorithm selects a bound B and constructs 2 squares by factorising many small numbers using a quadratic sieve. Quadratic sieve provides pairs of numbers with one number being a square and another having known factorisation. The sieve needs to find more than B numbers with all prime factors below B.

Then the factorisations are presented as vectors. Each vector entry stands for a power of a prime below B in prime factorisation of a number. Vectors are reduced mod 2 and placed into a matrix. The null space (linear combination of vectors which result in 0 vector) of the matrix is found, and it exists if number of vectors is above B, so at least 2 vectors have to be linearly dependent, creating the null space. It allows to construct $x^2 = y^2$ as a 0 vector mod 2 signifies a square (all powers of a square are even i.e. 0 mod 2), and vector addition represents multiplying numbers, as powers add during multiplication.

It is a subexponential algorithm like ECM but it is a general algorithm, meaning its running time depends only on the size of integer n rather than size of its prime factors. It runs heuristically in $\exp((1 + o(1)\sqrt{\log n \log\log n})$ time.

2.1.6 General Number Field Sieve

General Number Field Sieve (GNFS) or Number Field Sieve (NFS) is the current state of the art algorithm used for setting factorisation records and introduced in 1990 [21]. It is further improvement of QS by sieving numbers in Algebraic fields rather than $\mathbb{Z}/n$. The algorithm is currently has the best heuristic complexity, and is the fastest in practice for large enough integers. For example, already mentioned factorisation record set in 2020 was accomplished with the said algorithm [2].

At the same time the algorithm needs to factorise, just like QS, many small numbers which can be done using other factorisation methods like ECM. One of the implementations of the GNFS moves entire stage of the algorithm to a GPU and uses combination of Trial Division, P-1, and ECM to factor numbers [22] during the intermediate step.

The algorithm is very complex and requires many other well-tuned algorithms, so it will not be implemented in this paper.

2.2 Existing Benchmarks

Similar projects which access and compare existing algorithms have been attempted before. Paper introduces benchmark to compare wide range of algorithms [23]. The paper has many algorithms: trial division, Fermat's factorisation method, Pollard's rho, Pollard's p-1, William's p+1, NFS, QNFS, ECM, Dixon's method, Shanks square forms factorization, providing good comparison of results. The shortcomings of the paper is that it does not give details of the implementations of the benchmarked algorithms which may have affected the results like the method to select bounds or implementation of big integer arithmetic. For example, ECM showed low performance in comparison with other algorithms which could've been due to omitted implementation details.

Another paper also benchmarks the algorithms on both CPU and GPU, while comparing the results. However, it shows limited usefulness as it lacks many advanced algorithms like GNFS or ECM and provides the results only for small integers below 30 bits, limiting its usefulness [24].

It demonstrates the lack of the papers which provide benchmarks on modern hardware, comparing the performance of different algorithms. Many papers which implement 1 algorithm provide benchmark data but it can be hard to compare such results between each other as it maybe unclear how to account for differences in used hardware, especially if papers are written with significant time gap.

2.3 Factorisation Software

Integer factorisation is often implemented as a function in mathematical software, so the most popular mathematical software with factorisation capabilities is presented.

Mathematica is a very popular scientific software. The software is proprietary, so there is not information on exact implementation but the documentation claims that it uses a mix of trial division, Pollard's p-1, Pollard's Rho, ECM, and QS.

PARI/GP is an open-source algebra system, primarily oriented for the number-theoretic algorithms [25]. For factorisation it uses multi-polynomial quadratic sieve in combination with Shanks SQUFOF, Pollard's Rho, and ECM. It also verifies the result using a primality test. It is combination of the GP algebra language and PARI number theoretic library, which can be called separately from a C++ program.

ECM-GMP is another implementation of integer factorisation, which is probably the most famous ECM software. The implementation is highly customisable, allowing the user to select the bounds for stage 1 and stage 2, select number of curves to try for stage 1 and many other parameters. It is also part of **SageMath** mathematical library, providing more traditional interface, as well as default choice of parameters.

3 Methodology

Selected algorithms were implemented in several iterations, testing the performance at each stage. The implementations started with textbook versions of the algorithms programmed in Python, followed by porting the same textbook versions of the algorithms to C++. At C++ stage, several enhancements are tested and algorithms are parallelised. Then selected algorithms are accelerated with GPU using CUDA. In the end, 3 software implementations of factoring were also benchmarked to compare the achieved performance with the state of the art.

It was decided to measure the performance of the algorithms on semiprime numbers (numbers with 2 prime factors), to be able to easily control the size of the smallest factor, and be able to clearly state which algorithms have the best performance at which factor size.

Wall time was used throughout to access the performance instead of CPU clock time. The decision was made to be able to compare the performance across different hardware setups: single thread CPU, multithreaded CPU, and GPU, and the wall time provided a uniform way to measure performance. The downside of the decision was that performance of the algorithms can potentially be affected by the side effects like other applications consuming system resources, or overheating resulting in underclocking, but the best effort was made to account for those factors during the experiments.

A simple framework was developed at C++ stage to aid with benchmarking. It was used for generating the semiprime numbers to be factored, loading the test files, timing algorithms, and allowing manual input for testing.

Timings did not include I/O for file reading and writing. Prime number generation was also excluded from the benchmark with 1 table precomputed at program start and passed to algorithms if needed. The table was computed using the Sieve of Eratosthenes (appendix B). The output from algorithms is verified that it divides n and isn't 1 or n itself.

3.1 Python Implementations

Trial Division and Pollard's Rho were implemented as given in related work. Modular multiplication was implemented by performing multiplication followed by modulo n.

Pollard's p-1 algorithm was used with the starting bound 10000 as was suggested in [4]. Only stage 1 was implemented. In case n was returned, starting seed was chosen as next prime. In case no factor was found bound was multiplied by 100. Gcd was computed every 100 steps.

Elliptic Curve Method used Weierstrass parameterisation and only stage 1. The bound was chosen from the complexity analysis of ECM in [4, p. 338], which was $\exp((\frac{\sqrt{2}}{2} + o(1))\sqrt{\log p \log\log p})$, where $o(1)$ was approximated as 0 and p was set to $\sqrt{n}$ as roughly the size of a factor of a semiprime.

Quadratic Sieve used the $f(x) = x^2 - n$ polynomial for sieving, starting at $x = \sqrt{n}$. The matrix was stored in explicit form using Python lists, which was the main source of the inefficiency as it required $\pi(B_1)^2$ space ($\pi(x)$ is prime counting function). Then null space of the matrix in $\mathbb{F}_2$ field was computed with the **Galois** package, and relations were multiplied together to obtain a square.

The timing was done using the built-in Python **time** module.

3.1.1 Python Results

The most efficient algorithm implement in Python was Pollard's Rho, and least efficient Quadratic Sieve due to its naive implementation of the matrix.

It was decided that quadratic sieve and Pollard's p-1 will not be implemented in C++. Pollard's p-1 basic version would fail even for large bounds around 1,000,000 on integers starting from 60 bits. Given that it has a better and more interesting alternative ECM, which can be parallelised, it was decided to not proceed with the algorithm. Meanwhile, QS would require a lot of tuning for intermediate steps, and it will be more beneficial to spend time on the remaining algorithms to make sure they have implementations which accurately represent their performance.

3.2 C++ Implementations

The initial algorithm implementations directly copied the ones from Python. The development was done on the **Windows Subsystems for Linux (WSL)**, and the code was compiled with g++.

3.2.1 Big Integer Arithmetic Library

The final choice for the Big Integer Arithmetic was the famous **GNU Multi Precision Arithmetic Library (GMP)**. The library implements dynamic size for the integers and is optimised for speed. The GMP uses inline assembly extensively with SIMD instructions.

The library also supports implementations of essential number theoretic functions like GCD and modular inverse.

GMP provides C and C++ interfaces, and almost always C one was used. It allowed to ensure no hidden memory allocations happen, as C interface requires all variables to be initialised before usage.

Initially, it was planned to develop the C++ implementations on Windows and use the **MPIR** library, a fork of the GMP, ported to Windows. However, the performance of the arithmetic was extremely low, even worse than Python. It motivated the switch to the original GMP library and the target operating system to Linux as GMP was written for the Linux build chain.

3.3 Montgomery Multiplication

Some implementations tried using **Montgomery multiplication**, an optimised version of the modular multiplication [26]. It works well, when many multiplications are performed with the same module.

The Montgomery multiplication used Montgomery space, which is simply all integers multiplied by a constant $r > n$ (mod n). r must be a power of 2 for the algorithm to be efficient, and $\gcd(r, n) == 1$. It can be easily seen that addition and subtraction are the same as in normal space. Let $*$ be multiplication in Montgomery space, $ar + br \equiv (a + b)r$ (mod n). At the same time multiplication is different as $a * b = arbr = abr^2$ (mod n), so to keep the result consistent, it must be multiplied by r^{-1} to get abr. The fact that r is power of 2 allows to perform the Montgomery reduction efficiently i.e. finding xr^-1 (mod n).

Montgomery Reduction requires setup for specific modulus by setting 2 constants r and $\overline{n}$.

After that the algorithm can be performed like this for some integer u.

Algorithm 4 Montgomery Reduction

```
u_lower ← u **and** r_mask
u_lower ← u_lower * $\overline{n}$
u_lower ← u **and** r_mask
u_lower ← u_lower * mod
u_lower ← u_lower * u
u ← u_lower shift right by r bit number
**if** u > mod **then**
    u ← u - mod
**end if**
```

So reduction can be done using 3 multiplications, 2 bitwise ANDs and a bitwise shift.

Numbers are brought into Montgomery Space with 1 normal modular multiplication $a \equiv ar$ (mod n), resulting in overhead. To get number out of Montgomery space, the already established reduction is performed on ar, $arr^{-1} = a$.

The arithmetic was implemented on top of GMP library. There was added a precompiler directive flag which can be set to 1 or 0, selecting the implementation of the modular multiplication function. It can utilised Montgomery arithmetic or perform a normal multiplication followed by mod operation.

3.4 Multithreading

Multithreading was done using the C++ standard **thread** library. To synchronise algorithms, a volatile boolean flag was passed to each thread, which was checked at each iteration of the main loop. If it was set to true, algorithm would terminate as it would signify a factor is found in a different thread.

3.4.1 Benchmarking Framework

All further experiments were done with a simple benchmarking framework for the ease of algorithm comparison. The benchmark uses command line interface with arguments implemented with **cxxopts** library for parsing. The framework contains 3 executables:

- **factorise_integers:** It runs the benchmark, recording the timing results in factorisation_results.csv file
- **test:** runs the unit tests
- **generate_benchmark:** generates a benchmark file rsa_numbers.csv according to the parameters arguments

All implemented command line arguments can be found in the appendix C. The framework also provides a README.md file with all the dependencies used in the code.

Mean and standard deviation are appended to the timing results. Framework also has support for the Interrupt Signal (Ctrl+C, SIGINT) and calculates the statistics for the unfinished benchmark in case of manual program termination.

The time execution of each algorithm was measured with the the standard **chrono** library, using high_resolution_clock function, which uses the most accurate clock available on the system.

The framework along all the algorithms was build using a makefile.

3.4.2 Trial Division

Trial Division did not undergo any improvements besides utilising multiple threads.

Trial division was parallelised by splitting the range $[2, \lfloor\sqrt{n}\rfloor]$ where possible factor can be present into $l = \lfloor\sqrt{n}\rfloor/t$, $r_i = [2 + i * l, (i + 1) * l]$ for $i \in [0, t - 2]$ and $r_{t-1} = [(t - 1) * l, \lfloor\sqrt{n}\rfloor]$ for t threads. Each thread i performed the trials only on the region r_i.

3.4.3 Pollard's Rho

Two algorithm enhancements were tried besides parallelisation. First, Montgomery multiplication was used for the squaring inside the polynomial. It suits this algorithm particularly well as the modulo and variable are unchanged. On top of that, multiplication by r (mod n) is a bijection because $\gcd(r, n) = 1$, so even initial transformations does not have to be performed, as random starting point in $\mathbb{Z}/n$ will be chosen with the same probability as ar. Transformation out of the space also never has to be performed because factor is found with a gcd operation which holds in the Montgomery space. So Montgomery arithmetic can be implemented with minimum overhead.

A practical improvement of the algorithm can be accumulating the differences by multiplying them together into a product mod n. In the end gcd of the product can be taken as gcds hold under modular multiplication (lemma D.1), so if one difference produces a factor then the product will do as well, and the expensive GCD operation does not have to be applied every step.

Second improvement consisted in accumulating the differences inside a product mod n, so the expensive gcd operation does not have to be performed every step. It works as gcd mod n is multiplicative (if $\gcd(a, n) = \overline{a}$ and $\gcd(b, n) = \overline{b}$ then $\gcd(ab, n) = \overline{a}\overline{b}$).

The gcd was tried to be run at 10, 100, and 1000 steps, and 100 steps were chosen in the end as 1000 steps would result in a high chance of producing n as result of the gcd for small integer sizes like 30 and 40. It happens when between gcd checks 2 differences were multipled which contained different factors of n.

The algorithm was parallelised by choosing a random starting point point u_0. Constant for polynomial was fixed at 1.

3.4.4 ECM

First naive implementation uses Weierstrass curves. Point multiplication exactly followed the provided formulas and scalar multiplication was done using the classical recursive power ladder algorithm.

There point is current value, and point_original is saved value of original point.

Algorithm 5 Power Ladder

```
INPUT: multiple, point, point_original
if multiple = 0 or multiple = 1 then
    return
end if
is_odd ← multiple mod 2 = 0
multiple ← multiple / 2
MultiplyPoints(point, multiple)
AddPoints(point, point)
if is_odd then
    AddPoints(point, point_copy)
end if
```

The algorithm was parallelised with t threads by trying t curves in parallel.

We can improve the algorithm using the stage 2. The initial bound B_2 for stage 2 would be $B_2 = 100B_1$ as recommended in [4, 343].

The stage 2 for ECM follows closely the version for Pollard's p-1 and handles the same case for order of a point - when it has 1 large prime factor between B_1 and B_2. The classical stage 2 for the Weierstrass curves starts with $Q = [M]P$ iterates over primes in $(B_1, B_2]$ and computes $[M + p_i]Q$, which can efficiently be done by multiplying point by a difference $\Delta_i = p_i - p_{i-1}$, so $[p_i]Q = [\Delta_i + p_{i-1}]Q = [p_{i-1}]Q + [\Delta_i]Q$. As differences between prime numbers are usually small and even, $[\Delta_i]Q$ can be stored in a table a reused. It results in testing 1 prime in stage 2 requiring only 1 addition on the Elliptic Curve.

The second improvement to the algorithm can be made with the use of Montgomery curves as described in the related work. The multiplication is a bit more difficult due to the need of difference between points. The code was adapted from the algorithm given in [27, p. 332]. The implemented version was altered as there was seemingly a typo making additions after the check for 0$^{\text{th}}$ bit incorrect.

The convention below is that last point in AddPoints is the difference between the first 2.

The Montgomery version of the algorithm used the special parametirisation with sigma from [15].

Montgomery curve parametirisation has its own version of the 2$^{\text{nd}}$ stage, which allows to reduce multiplication by a difference from 1 elliptic curve operation to 2 modular multiplications by storing points for differences between prime numbers [4, p. 342].

The bounds can also be tuned specifically for the algorithm as discussed in the [13]. First the optimisation has a parameter K which specifies how faster stage 2 is per unit of bound. Timing the algorithm showed that K=27 for this implementation. So time can be defined as $T = B_1 + \frac{B_2 - B_1}{K}$, and then we can maximise the function $\frac{P(\alpha,\beta)}{T}$.

A Python script was used for the optimisation and the results were stored in a file, with starting at p=10 bits and ending with p=108. To optimise the function, it was sampled at (b_1, b_2) iteratively in the in the intervals $b_1 \in [2, 2^p/2]$ $b_2 \in [3, 2^p/3]$, by multiplying first variables by

Algorithm 6 Scalar Multiplication On Montgomery Curves

```
INPUT: point, multiple
point_copy, point_double ← point
point_double ← DoublePoint(point_double)
c ← bit count of multiple
for bits $b_i$ of multiple with $i \in [c-2, 1]$ do
    if $b_i$=1 then
        point_copy ← AddPoints(point_double, point_copy, point)
        point_double ← DoublePoint(point_double)
    else
        point_double←AddPoints(point_copy, point_double, point)
        point_copy ← DoublePoint(point_copy)
    end if
end for
if $b_0$ = 1 then
    point ← AddPoints(point_double, point_copy, point)
else
    point ← DoublePoint(point_copy)
end if
```

1.1 and finding the maximum (m_1, m_2). The second round of optimisation tried improving the maximum by sampling $b_1 \in [m_1/2, m_1 * 2]$ and $b_2 \in [m_2/2, m_2 * 2]$, multiplying variables by 1.05 this time, for higher precision. To sample the function, SageMath was used for the Dickman's function and it integrated using **SciPy's** quad function.

Given that parametirisation makes the group divisible by 16, smaller bound sizes were sampled by choosing $\log_2 \sqrt{n} - 3$ bit size.

3.5 Generating Benchmarks

The benchmark was generated with 2 functions, the first one to find a prime within set range, and the second determined the ranges to obtain a semiprime within required size. The benchmark generation is seeded as well to be able to generate the same benchmark by just storing the parameters rather than the entire file.

The first RandomPrime function is implemented by picking random number and testing it for primality. It avoids bias towards a particular type of prime numbers. The arguments are lower_bound, upper_bound.

Algorithm 7 Generate Prime

```
random_bound = upper_bound-lower_bound
while true do
    random_number ← Random(random_bound)
    random_number ← random_number + lower_bound
    is_prime ← Repeat Primality Test 50 times
    if is_prime then
        return random_number
    end if
end while
```

The prime test uses GMP implementation that starts with trial divisions, then 24 times Baillie-PSW test followed by 26 repetitions of Rabin Primality test for 50 repetitions. According to documentation it will falsely detect a composite number as prime with probability of 4^{-r} for r repetitions (r=50 in this case).

It selects the first factor randomly, however it is biased to be big, as there are the same amount of numbers in $[2, 2^{p-1}]$ and in $[2^{p-1}, 2^p]$.

And below is the pseudo-code for making the product of 2 primes to be of required size i.e. $2^{bit\_size-2} < n < 2^{bit\_size+2}$:

Algorithm 8 Generate Semiprime

```
p1 ← $\mathrm{RandomPrime}(2, 2^{\lfloor \mathrm{bit\_size}/2 \rfloor + 1})$
p1_size ← $\lfloor \log_2(\mathrm{p1}) \rfloor$
p2_size ← bit_size − p1_size
p2 ← $\mathrm{RandomPrime}(2^{\mathrm{p2\_size}-1}, 2^{\mathrm{p2\_size}})$
```

3.6 Testing

Due to the probabilistic nature of the algorithms, errors in the number theoretic functions result not in an obvious error or a crash, but rather in decrease in the performance. It motivated spending additional time on inclusion of unit tests to ensure performance results are accurate and are not affected by code errors.

The **doctest** framework for C++ unit tests was chosen due to small scale of the project and ease of integration - as it is enough to just include one header file. It provides useful statistic on the number of the failed assertions as well as outputting the compared values in case of a failed assertion.

The tests were written for the Montgomery Arithmetic, Elliptic Curve Arithmetic, Prime Generation, and Benchmark Generation. The tests were probabilistic due to the difficulty to consider all edge cases by hand (and if it was easy, the unit tests were not as necessary in the first place). The tested numbers as well as the sizes of the tested integers were randomised.

Montgomery arithmetic was tested in 4 ways:

1) Transfer number into and out of the Montgomery space and verify that the number is the same.
2) Transfer 2 numbers into Montgomery space, perform addition or subtraction there and same operations outside of the Montgomery Space. Check that numbers are the same after results taken out of the Montgomery Space
3) Same as above, but perform multiplication in the Montgomery Space
4) Transfer a number into Montgomery space and verify that GCD with n equals to the GCD outside the space

Elliptic Curve Operations were tested to be Abelian. Weierstrass curves and Montgomery curves were tested in similar way, multiplication function was used to generate 3 points from 1 randomly selected initial point with multiplication such that [n]P + [m]Q = [a]R with n + m = a for Weierstrass Curves, and 4 points were Generated for Montgomery Curves to also be able to perform 1 addition operation. The test verifies that implemented curves indeed implement Abelian Group operations. The test was repeated 10,000 times for both Montgomery and Weierstrass curves to hopefully catch edge cases if they were present.

To test benchmark generation, 1 test verified that primes outputted by GeneratePrime function were indeed prime, using GMP primality test, and the other test checked that GenerateSemiprime produces numbers within 2 bits of the required size.

3.7 CUDA Implementations

Trial Division, Pollard's Rho, and ECM were also ported onto GPU using CUDA. This step was done in the end, to be able to use the performance data and experience gained at previous stage to make judgement about which version of the algorithms would be implemented.

3.7.1 CUDA Programming Model

GPU is Single Instruction Multiple Thread (SIMT) hardware. It uses threads split into warps of 32 each, and only single instruction can be executed within a warp by all threads. So, if several threads take different path on a branch, some of them will stall. Threads are also combined into larger blocks, which can contain at most 512 threads [28].

Memory is split into shared memory which is available to all threads within same block. Global memory is available to all threads within the same block (fig 3).

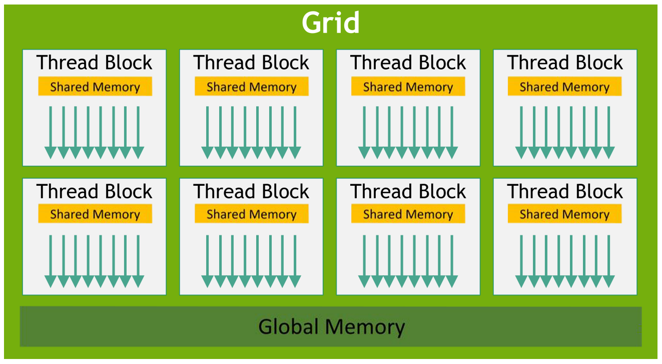


Fig. 3. Diagram of CUDA threads and memory, edited from [28]

All data has to be copied to the GPU before it is made available, which may result in significant overhead if data is being frequently requested by the CPU, or jobs with new data are scheduled.

It is also important to note that GPU tend to have much higher core count than CPUs but lower clock frequency, which result in slower execution of serial code.

To synchronize the algorithms, a volatile boolean flag was used again. On GPU it was stored in global memory.

3.7.2 Big Integer Arithmetic on GPU

The **Cooperative Groups Big Numbers (CGBN)** library will be used instead of GMP on the GPU. It is developed by NVIDIA, and provides functions for big unsigned integer arithmetic on GPU.

The primary reason the library was selected is that it focuses on speed with the claimed speed up of more than 100 times for addition and 28 times for Montgomery multiplication. It achieves such large improvement due to using several GPU threads to handle 1 integer computation, which allows to utilise high warp synchronisation and use more registers during integer operations, saving time on memory accesses. So the library allows to perform computation on many problem instances simultaneously while dedicating several threads per instance of work.

The main downside of the library is that it uses fixed-integer arithmetic which has to be chosen during compile time. So, the integer speed would be sub-optimal for almost all integer sizes as it has to be set to fit the largest integers. A potential solution for that would be to compile CUDA algorithms as separate executables, which can be recompiled for every bit size, however this idea was not performed due to the schedule constraints.

The library is also allows to choose how many threads would be used simultaneously for 1 arithmetic operation. The optimal size would be 32 as it would ensure that entire warp works together, however many algorithms did not require such size, so there will be no work for all of the threads. The number of threads was set to be the number of 32 bit limbs used in the fixed size arithmetic representation.

The library also has been unmaintained for 2 years and had two issues which had to be fixed. The samples and unit tests did not compile due to the outdated makefile which used master branch name instead of main. And utility functions were defined in header files resulting in linking error when included in several files which are linked together. The first error was fixed by editing the makefile and the second error by marking the utility functions as inline.

3.7.3 Trial Division

The advantage of the trial division is that every trial is independent which allows for great parallelisation. As only modular operation is performed, then the range for integer size has to be only n.

The implementation followed the C++ multithreaded parallelisation by first splitting the range into roughly equal sub-ranges and assigning each to a problem instance.

The algorithm was compiled with 128 bit integer size and 4 threads per operation.

3.7.4 Pollard's Rho

The algorithm was implemented by passing an array of starting values and values a for the polynomial $f(x) = x^2 + a$. The algorithm was tried using usual modular multiplication and the Montgomery multiplication as it was decided that CGBN implementation might be more efficient than the code written for the paper.

Pollard's Rho would unfortunately require 2n space for integers to store upper bits of multiplication.

The algorithm was compiled with 256 bit integer size and 8 threads per operation.

3.7.5 ECM

It was decided to parallelise 1 stage algorithm with Montgomery parallelisation as it looked the most promising, due to windowing of stage 2 would have likely be less performant on the GPU due to lack of available memory for GPU threads as was suggested in [18].

The algorithm was compiled with 512 bit integer size and 16 threads per operation.

3.8 Measuring Performance of the Factorisation Software

3.8.1 Mathematica

Mathematica's FactorInteger function was benchmarked by writing a Mathematica notebook. It did the following:

1) Read the semiprimes for a particular bit size from a file into an array
2) Factorise the semiprimes and store result into another array, timing the execution time with the AbsoluteTiming function
3) Array is written into a file

The output had to be manually formatted to be the same as the framework output. The factorisation was measured with the AbsoluteTiming function as it measures the wall time, rather than CPU time measured by the Timing. It makes sure the timing is consistent with including the setup overhead of the remaining algorithms. The Mathematica, unlike other software, was installed and was benchmarked on Windows.

3.8.2 PARI/GP

PARI/GP provides its algebra system PARI as a C library. It was integrated into the factorisation framework as an another algorithm, with a wrapper which converts GMP integers to the PARI format GEN, calls Z_factor(), and converts the output back to the GMP format.

3.8.3 GMP-ECM

GMP-ECM programmable interface was implemented in already utilised **SageMath** package. So, it is added as a function into the Python code and factor size is provided to choose the bounds for ECM.

4 Results

The tables present the results taken as average of the factorised numbers for the particular bit size.

4.1 Setup

4.1.1 System and Hardware

All code was run on the following hardware:

- **CPU:** Intel Core i7-10875H
- **GPU:** NVIDIA GeForce RTX 2080
- **RAM:** 32.0 GB
- **SSD:** 512 GB Samsung MZVLB512HBJQ-0000

The final results for C++ and CUDA are from the benchmarks ran on **Debian** Linux distribution to avoid potential overhead from virtualisation of WSL. The CPU clock frequency was set to be between 2.1 Ghz and 2.3Ghz using **cpupower** utility to prevent the CPU from overheating and subsequently underclocking during the benchmark, as it would have affected the wall time. The thread number was limited to 8 to ensure other processes do not compete with the benchmark for computation resources and have less impact on performance.

Python was ran on Windows as it initially the development was for this platform and code was configured to work with it.

As Pari was integrated into the C++ framework it was benchmarked on Debian as well. Mathematica was benchmarked on Windows. GMP-ECM was benchmarked on Windows as it was integrated into the Python code with SageMath.

4.1.2 Compilation and Code Execution

The compilation was done using the -Ofast setting as it prioritises execution speed of the code. Further, link time optimisation was turned on with the -flto flag to enable the compiler to optimise the code using the context of several object files. NVCC was used to compile the CUDA code with O3 flag (maximal optimisation possible) for the host code, and the GPU code is optimised by default.

For Python the standard CPython version was used.

4.1.3 Final Semiprime Benchmark

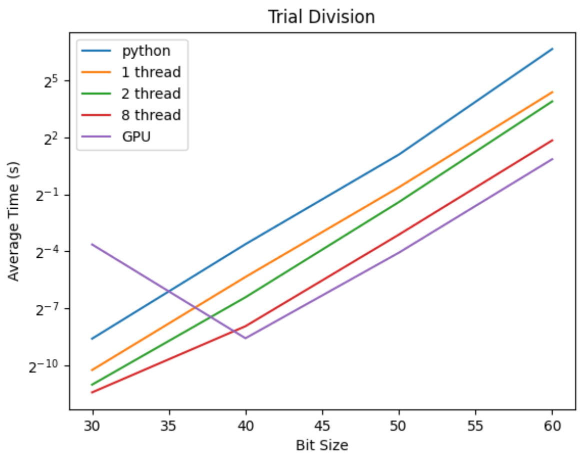
The final benchmark uses 30 numbers for each bit size. Initially, 10 numbers were thought to be sufficient but it was increased to 30 due to high variance of the averages of the ECM even for exactly the same code. For example averages for the 1 stage Montgomery implementation of the algorithm for 150 bit integers: 3.81933, 7.40841, 8.50289, 12.7623, 11.7161, 4.79651, 5.19744, which shows up to almost 3 time difference between averages. 30 was chosen as a balance between consistency of the results and reasonable benchmarking time for each bit size. The same benchmark was used for all algorithms with the **seed: 815430918**.

4.2 Result Graphs

4.2.1 Trial Division

The presented benchmark for trial division presents implementations for Python, C++ with 1, 2, and 8 threads and GPU with 2048 problem instances (fig 4).

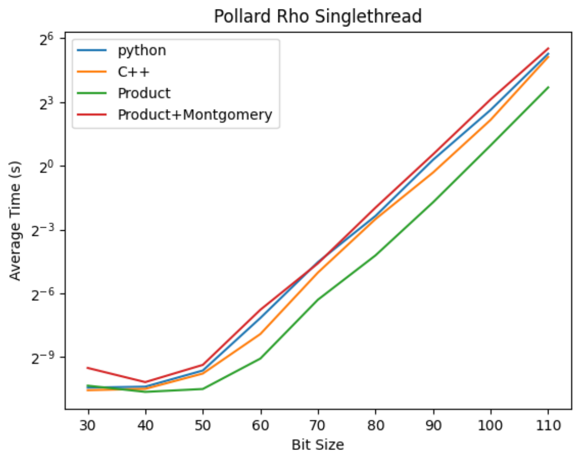
Fig. 4. Trial Division Performance Benchmark



4.2.2 Pollard's Rho

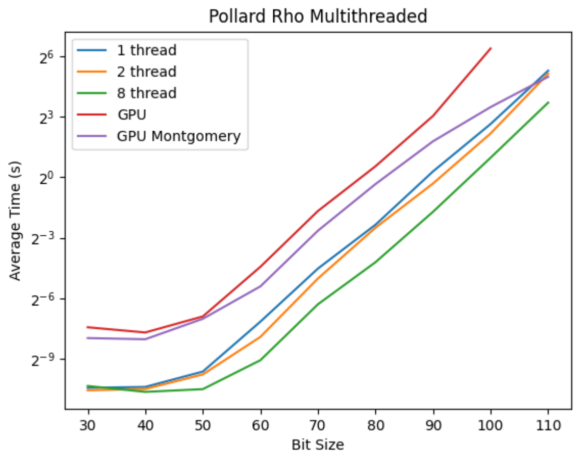
The Pollard's Rho benchmark for single threaded algorithm presents python, C++ implementations, and 1 enhancement by accumulating differences into a product, and another enhancement on top of previous one of using Montgomery Arithmetic (fig 5).

Fig. 5. Pollard's Rho Single Thread Performance Benchmark



The best single threaded used just product accumulation, so it was multithreaded. CUDA version used 2048 problem instances as well (fig 6).

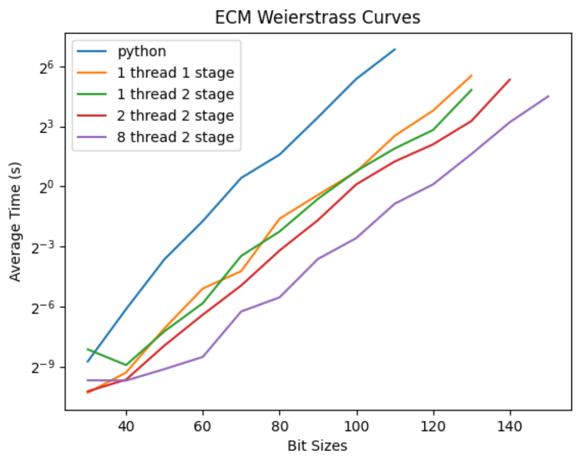
Fig. 6. Pollard's Rho multithreaded Performance Benchmark



4.2.3 ECM

First, the Weierstrass curves were benchmarked, stating from stage 1 and then adding stage 2. As performance was better for stage 2, it was tested in multiple threads (fig 7).

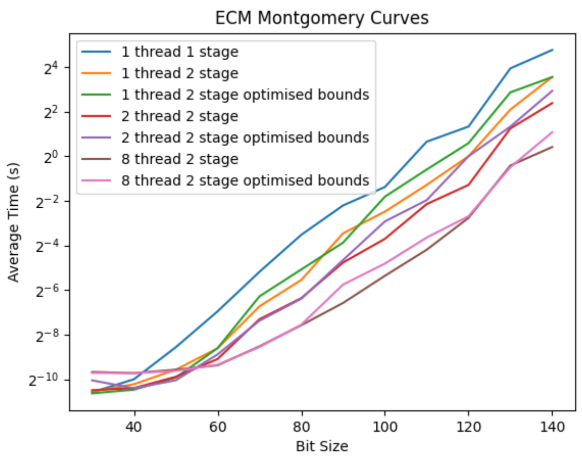
Fig. 7. ECM Weierstrass Parameterisation Benchmark



Montgomery curves also were tried with and without stage 2. As stage 2 improved performance, better bounds optimised according to [13] were tried, which resulted in slightly worse performance. To ensure the same behaviour, both default and optimised bounds were tried with 2 and 8 threads, with default bounds showing better result in all cases (fig 8).

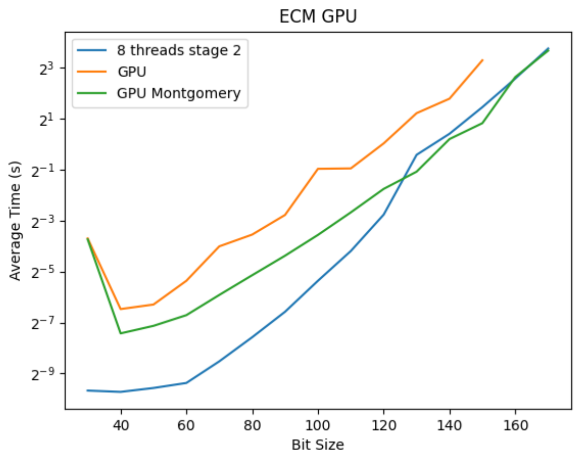
As Montgomery Multiplication showed worse performance with Pollard's Rho, it was not used for ECM.

Fig. 8. ECM Montgomery Parameterisation Benchmark



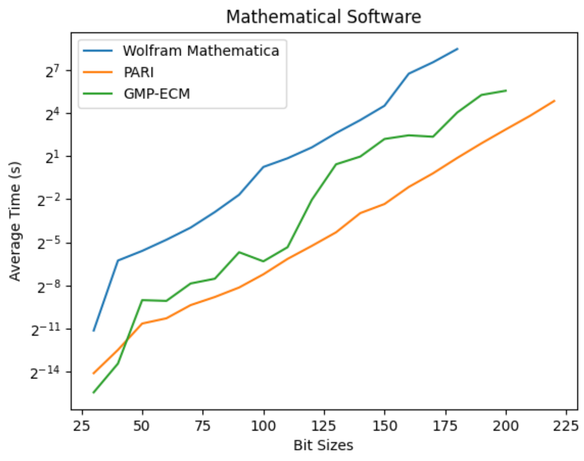
The stage 1 algorithm with Montgomery Curves is compared with GPU implementations with and without Montgomery Arithmetic. 512 problem instances were used this time (fig 9).

Fig. 9. ECM GPU Benchmark



4.3 Factorisation Software

Fig. 10. Mathematical Software Benchmark



5 Evaluation

The benchmarks showed mix of expected and unexpected results. The best algorithm for larger integers was multi-threaded ECM with 2 stages

The factorisation software did demonstrate that algorithmic implementations can in theory provide much greater performance than hardware improvements. The best implemented algorithm - 2 stage ECM on 8 threads did perform comparably with the state of the art GMP-ECM, however PARI library provided way greater speed even with 1 thread in comparison with 8 used by the implemented version of ECM.

5.1 Algorithm Implementations

The algorithm implementations were varied, but some of them clearly did not represent well the actual performance. There were also a decent number of algorithm variations left.

For example, Montgomery multiplication performed worse than usual modular multiplication, while CGBN implementation was other way around providing performance improvements. The discrepancy was likely due to an oversight, noticed only after running the benchmarks, that the implementation used 3 multiplications instead of 2.

Another enhancement which did not provide benefit was the bound optimisation for ECM, which performed consistently slightly worse than the default bounds, however it was closed. An issue can be in regards to the used K value, which was 27, while [13] claims that the best algorithms have it at 100 or above, which likely signifies inefficiencies in the stage 2 implementations.

Pollard's Rho had slight enhancements, but multithreading was the only enhancements which considerable improved performance.

5.2 Evaluation of the GPU Performance

GPU showed that it can improve the performance if algorithm scales well from large number of threads, benefiting trial division.

Trial division was the only algorithm which had consistent performance improvement of 2 times over multithreaded version, showing that the problem lies in the way algorithms are ported or GPU architecture rather than integer arithmetic itself.

ECM on GPU was the main disappointment as it did not improve despite predicted and observed linear scaling with number of threads. The most likely reason for it is that 512 curves tried simultaneously was too many as runs for integer of 140 bits would not always exceed even 100 curves, resulting in wasted computation. For example, other papers which implemented algorithms on GPU like [18] and [22] used GPU to maximise throughput rather than time to factorise individual integers, and reported high improvements in performance in comparison with CPU.

Another reason for GPU poor performance in the case of factorisation of individual RSA integers, maybe be too low integer size. It can be seen that initially GPU for ECM is way slower than multithreaded implementation, but with increase in integer size, it becomes closer and sometimes overcomes it.

In conclusion, GPU is best used when the main requirement is throughput rather than speed of individual factorisations. And further benchmarks are required to check if at some point GPU version would overcome multithreaded one in speed.

5.3 Comparison with the Sate of the Art

It should be noted that the benchmarked algorithms do not use multi-threading, so the comparison is not entirely fair (and GMP-ECM is capable of running on multiple threads or on a GPU). Still, it is demonstrative that higher hardware utilisation can result in a speed greater than the best ECM implementation. As it can be seen that 8-threaded ECM with Montgomery parametirisation performed poorer on small integer sizes up to 110 bits and started to perform better after that threshold.

However, an implementation which does not limit itself in used algorithm, PARI, produces faster factorisation than even 8 threaded ECM.

It signifies the importance of both hardware utilisation and use of correct algorithm for the circumstances. As a general algorithm - QS was predicted to be better than specialised one like ECM for semiprimes with large factors.

TABLE 1
Comparison between 8 threaded ECM, GMP-ECM, and PARI

	ECM 8	GMP-ECM	PARI
30	0.00122726	2.26E-05	5.68E-05
40	0.00118429	9.04E-05	0.000175
50	0.00131672	0.001925	0.000623
60	0.00150956	0.001855	0.000801
70	0.00271516	0.004295	0.001514
80	0.00523395	0.00541	0.002241
90	0.010465	0.019376	0.003537
100	0.0243225	0.01249	0.006681
110	0.0546611	0.024708	0.014051
120	0.147135	0.242464	0.026298
130	0.147135	1.343187	0.050613
140	1.32715	1.952586	0.127724
150	2.73823	4.55533	0.198059
160	5.95865	5.432916884	0.450227
170	13.4937	5.073797085	0.869185

5.4 Benchmarking Accuracy

The integer count was sufficient for all the algorithms besides ECM, where random pattern is strongly visible on the graph, despite already increased integer size.

Arguably, the main problem with benchmark was that the clock speed of GPU was not controlled, It might have affected the GPU with first overclocking and then underclocking, making results less stable.

Benchmarking can be further improved by raising number of integers per bit size to 50 or 100 integers, however it would have made the experimentation more difficult. The optimal solutions would have been to use 30 integers per bit size during development, and in the very end utilise 100 bits for the final benchmark to ensure reliability of the results.

Some graphs also show unreasonable inefficiency for GPU algorithms at 30 bit size. The problem was not investigated due to time constraints and expected behaviour for other integer sizes.

5.5 Benchmarking Methodology

Benchmarking was done only on the entire algorithms, while it could have been beneficial to carefully access performance of individual functions like point addition, modular ,multiplication, speed of going through a curve at a fixed bound. It could have accelerated the development by simplifying performance assessment, and making fine-tuning individual functions easier.

6 Conclusion

6.1 Accomplishment of Deliverables

The project has fully completed all basic deliverables, implementing selected algorithms in a textbook way. The intermediate and advanced deliverables were mostly completed, however further work would have been appropriate.

In relation to intermediate deliverables, some important enhancements on ECM were missing like Edwards curves or using windowing for scalar multiplication. On the upside, 3 algorithms were successfully parallelised, achieving significant performance increase.

The advanced deliverables were also partially completed with successful GPU implementations. However, it is not possible to say definitively if there is potential for GPU implementations to outperform CPU multithreaded algorithms for factorisation of 1 integer. Benchmarks for larger integer sizes and testing other versions of the algorithms would be appropriate.

6.2 Work Significance

The differentiating point of the paper is that it performed extensive benchmarking of several algorithms. Other papers, concentrating on benchmarks would either perform factorisation either in small integer ranges or on small sample sizes, leaving performance of many algorithms up to a chance. This paper both considered integers up to 140 bits for the implementations and samples of 30 integers which mostly removed randomness from the results. We have also taken extra care to make sure benchmarks are unaffected by CPU clock speed or virtualisation.

The framework is also easy to use, so it would be easy to test the same algorithms on different hardware to provide a baseline for performance in future research.

The benchmarks also answered the question which improvements algorithmic or implementation were the most significant for the performance. The results showed that choosing appropriate algorithm will have the most significant impact on the example of PARI, but if algorithm version is fixed then both hardware and software would be appropriate. So, it would be reasonable for factorisation implementation to first concentrate on finding the best algorithm and then try to fine-tune it for particular hardware.

6.3 Further Work

6.3.1 Implementation Improvements

Currently, the arithmetic is sub-optimal for both the CPU and GPU. GMP uses dynamic range, which did not utilise the assumption that we are working in the ring $\mathbb{Z}/n$, so all the integers have limited range and additional checks are not required. GMP also uses dynamic memory allocations for each integer, which creates overhead on its own, and potentially makes memory more fragmented, which results in more cache misses. Using fixed integer arithmetic can alleviate this problem by allocating all data on the stack. Number Theory Library is a C++ library which can potentially be more performant than GMP in this case as it provides implementations for modular arithmetic. Another option would be a custom implementation done with GMP low level functions, which operate on arrays of limbs and do not provide any memory allocations of their own, allowing for greater control.

GPU uses fixed-size arithmetic, so the problem of dynamic memory is alleviated, but the size of the integers has to be known at compile time, which creates extra overhead. This can be fixed by recompiling the CUDA code for every bit size, and calling the executable as a separate process. This technique can provide some overhead, but also may be more efficient for integers below the maximal size, so it would be an interesting topic for further experimentation.

6.3.2 Further Algorithmic Improvements

Besides Edwards Curves for the ECM, Quadratic Sieve is a popular algorithm which can also be benchmarked in the same manner as was done with ECM. Potential variations may include selecting different bounds, trying other algorithms for finding null space of a matrix like block Wiedemann algorithm.

Several promising algorithms which involve FFTs also can be implemented. For example, the improved Pollard's Rho for multithreaded implementations [8] or using FFTs to accelerate stage 2 in ECM to bring number of multiplication down from 2 to 1 [9].

6.3.3 Other Benchmarking Scenarios

The paper has covered only 1 benchmarking scenario which involves worst case scenario for a particular integer size, but provides assumptions regarding factor size.

For example, ECM is often used as a part of GNFS implementation, to cofactorise many integers which may have more than 2 prime factors. This scenario would also prioritise throughput over speed on 1 integer, which will likely make GPU more efficient. So, the conclusion that GPU cannot provide significant improvement in performance is incomplete.

Another popular scenario is to factorise large integers like Fermat or Mersenne numbers, which often have special optimisations available only for numbers of such format. They are also very large in comparison with their factors, so efficient arithmetic would become even more important.

REFERENCES

[1] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[2] A. G. N. H. E. T. P. Z. Fabrice Boudot, Pierrick Gaudry, "The state of art in integer factoring and breaking public-key cryptogtaphy," *IEEE Security and Privacy Magazine*, 2022. [Online]. Available: https://hal.science/hal-03691141

[3] K. Rupp *et al.*, "years of microprocessor trend data, 2018," *URL https://www. karlrupp. net/2018/02/42-years-of-microprocessor-trend-data*, vol. 42, 42.

[4] R. Crandall and C. Pomerance, *Prime Numbers A Computational Perspective*, 2nd ed. Springer Science+Business Media, Inc., 2005.

[5] R. P. Brent, "Parallel algorithms for integer factorisation," *Number theory and cryptography*, vol. 154, pp. 26–37, 1990.

[6] J. M. Pollard, "A monte carlo method for factorization," *BIT Numerical Mathematics volume*, 1975. [Online]. Available: https://doi.org/10.1007/BF01933667

[7] A. K. K. Y. V. P. N. K. S. G. E. R. K. R. S. A. S. M. A. G. Harish, G. Punith Kumar, "Parallelization of pollard's rho integer factorization algorithm," *ACM*, 2012. [Online]. Available: https://dl.acm.org/doi/abs/10.1145/2393216.2393224

[8] R. E. Crandall, "Parallelization of pollard-rho factorization," *preprint*, vol. 23, 1999.

[9] P. L. Montgomery, "An fft extension of the elliptic curve method of factorization," Ph.D. dissertation, University of California, 1992.

[10] J. M. Pollard, "Theorems on factorization and primality testing," in *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 76, no. 3. Cambridge University Press, 1974, pp. 521–528.

[11] H. W. Lenstra Jr, "Factoring integers with elliptic curves," *Annals of mathematics*, pp. 649–673, 1987.

[12] P. Reid, "Part 2 — is elliptic curve cryptography (ecc) a step towards something more — understanding ecc," *Medium*, 2016.

[13]

[14] P. L. Montgomery, "Speeding the pollard and elliptic curve methods of factorization," *Mathematics of computation*, vol. 48, no. 177, pp. 243–264, 1987.

[15] R. Brent, R. Crandall, K. Dilcher, and C. Van Halewyn, "Three new factors of fermat numbers," *Mathematics of computation*, vol. 69, no. 231, pp. 1297–1304, 2000.

[16] H. Edwards, "A normal form for elliptic curves," *Bulletin of the American mathematical society*, vol. 44, no. 3, pp. 393–422, 2007.

[17] D. J. Bernstein and T. Lange, "Inverted edwards coordinates," in *Applied Algebra, Algebraic Algorithms and Error-Correcting Codes: 17th International Symposium, AAECC-17, Bangalore, India, December 16-20, 2007. Proceedings 17*. Springer, 2007, pp. 20–27.

[18] T. K. Joppe W. Bos, "Ecm at work," 2012. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-34961-4_29

[19] G. De Meulenaer, F. Gosset, G. M. De Dormale, and J.-J. Quisquater, "Integer factorization based on elliptic curve method: Towards better exploitation of reconfigurable hardware," in *15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2007)*. IEEE, 2007, pp. 197–206.

[20] C. Pomerance, "Analysis and comparison of some integer factoring algorithms," *Computational methods in number theory*, pp. 89–139, 1982.

[21] A. K. Lenstra and H. W. Lenstra, "The development of the number field sieve."

[22] T. K. A. K. L. Andrea Miele1, Joppe W. Bos, "Cofactorization on graphics processing units," *Cryptographic Hardware and Embedded Systems*, 2014. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-662-44709-3_19

[23] C.-L. Duta, L. Gheorghe, and N. Tapus, "Framework for evaluation and comparison of integer factorization algorithms," in *2016 SAI Computing Conference (SAI)*. IEEE, 2016, pp. 1047–1053.

[24] I. R. S. R. Kimsanova, G., "Comparative analysis of integer factorization algorithms using cpu and gpu," *MANAS Journal of Engineering*, 2017. [Online]. Available: https://dergipark.org.tr/en/pub/mjen/issue/40447/484351

[25] *PARI/GP version `2.15.5`*, The PARI Group, Univ. Bordeaux, 2024. [Online]. Available: http://pari.math.u-bordeaux.fr/

[26] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of computation*, vol. 44, no. 170, pp. 519–521, 1985.

[27] D. S. Dummit and R. M. Foote, *Abstract algebra*, 3rd ed. Wiley Hoboken, 2004.

[28] Official guide on CUDA programming by NVIDIA. [Online]. Available: https://docs.nvidia.com/cuda/cuda-c-programming-guide

[29] Lecture Notes for Durham Cryptography And Codes module from 2023-2024 academic year.

APPENDIX A
RSA ENCRYPTION AND FACTORISATION ATTACK

Let Alice wants to safely receive a message from Bob. Alice chooses e with $gcd(e, (p-1)(q-1)) = 1$, primes p and q, $n = pq$. Alice computes decryption exponent d with $ed = 1$ (mod (p-1)(q-1)) with extended euclidean algorithm. Alice keeps tuple (e, p, q, d) as a private key and sends (e, n) to Bob which is the public key. Bob encrypts message m encoded as an integer to cipher-text c $c = m^e$. Alice can obtain c from m as $m = c^{em}$ by Euler's Theorem.

Fast factorisation allows an attacker to obtain private key from the public key by factoring n into p and q, computing $(p-1)(q-1)$ and finding decryption exponent $d = (p-1)(q-1)/e$ from the private key and reversing the encryption [29, p. 20].

APPENDIX B
GENERATING PRIME TABLE

This algorithm allows to find all prime numbers in the range $[2, n]$ in $O(nloglog(n))$ time [4, p. 122]. It constructs a bit array of size n, initialised to 0. Then it starts with 2 and marks every 2nd bit as 1. Then it iterates from 2 until it finds first 0 at jth position. It marks every jth bit as 1 and continues moving until $\sqrt{n}$. After that it just stores position of every 0 bit as a prime until n. See pseudo-code below:

Algorithm 9 Prime Sieving

```
$primes \leftarrow$ empty list
$bound \leftarrow \sqrt{n}$
$bits \leftarrow$ array of 0's
**for** $2 \leq i \leq bound$ **do**
  **if** bits[i] = i **then**
    $primes$ append i
    $j \leftarrow i$
    **while** $j \leq n$ **do**
      $bits[j] \leftarrow 1$
      $j \leftarrow j + i$
    **end while**
  **end if**
  $i \leftarrow i + 1$
**end for**
**for** $bound < k \leq n$ **do**
  **if** $bits[k] = 1$ **then**
    $primes$ append k
  **end if**
**end for**
```

APPENDIX C
FRAMEWORK COMMAND LINE ARGUMENTS

The framework has 3 executables:

- **factorise_integers:** Runs the selected factorisation algorithm on the benchmark

  - **–algorithm -a** Selects the factorisation algorithm to use. Accepts full algorithm name or a short alias. Aliases:
    * trial_division (td)
    * trial_division_cuda (tdc)
    * pollards_rho (pr)
    * pollards_rho_cuda (prc)
    * pollards_p-1 (pp1)
    * ecm_weistrass (ecmw)
    * ecm_weistrass_2 (ecmw2) (Weierstrass parameterisation with stage 2)
    * ecm_montgomery (ecmm)
    * ecm_montgomery_2 (ecmm2) (Montgomery parameterisation with stage 2)
    * ecm_cuda (ecmc)
    * pari (PARI mathematical library)
  - **–manual -m** Turns on manual mode, which will allow to manually input integers
  - **–count -c** Count of the integers to be factorised before closing the program. Allows to run only the first of the benchmark
  - **–seed -s** Sets the random seed. Default: Sets to system time
  - **–thread_number -t** Sets number of CPU threads or GPU problem instances

- **generate_benchmark:** Generates a new benchmark with semiprimes
  - **–max_size -m** Sets the size up to which semiprimes will be generated. Default: 10
  - **–numbers_per_size -n** How many semiprimes should be generated per each bit size. Default: 10
  - **–seed -s** Sets the random seed for the generation - same seed will result in the same benchmark generated. Default: Sets to system time
  - **–initial_size -i** The semiprime integer size to start with. Default: 30
  - **–step -t** Size of the step when generating random integers. Default: 20

- **test:** Runs the randomised unit tests. Has no arguments