FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Parallel factorization on GPU using CUDA and Metal APIs
Student:	Bc. Jan-Jakub Fleišer
Supervisor:	doc. Ing. Ivan Šimeček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Systems and Networks
Department:	Department of Computer Systems
Validity:	until the end of summer semester 2023/2024

Instructions

1) Implement Pollard's rho and Lenstra elliptic-curve factorization algorithms[1-4] with arbitrary integer precision in their sequential, parallel on CPU using OpenMP, and parallel on GPU using Apple Metal and CUDA versions.
2) For sequential and CPU parallel versions, utilize the GMP library.
3) For the Metal GPU version, implement an arbitrary precision library with operations required for the algorithms.
4) For the CUDA GPU version, explore existing solutions for arbitrary precision integer arithmetic, and if required, implement a similar library as in the Metal version.
5) Compare the implementations created with widely used SageMath and SymPy solutions in terms of scaling, resource usage, and time until a solution is found.
6) Compare the resource usage and runtime for the implemented algorithms, both with each other and between their CPU and GPU versions.

[1] LENSTRA JR, Hendrik W. Factoring integers with elliptic curves. Annals of mathematics. 1987, pp. 649 673.
[2] PARKER, DANIEL. Elliptic curves and Lenstra's factorization algorithm. University of Chicago: REU. 2014, vol. 2014.
[3] BRENT, Richard P. Some Parallel Algorithms for Integer Factorisation. In: AMESTOY, Patrick; BERGER, Philippe; DAYDE, Michel; RUIZ, Daniel; DUFF, Iain; FRAYSSE, Valérie;GIRAUD, Luc (eds.). Euro-Par'99 Parallel Processing. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 1-22. ISBN 978-3-540-48311-3.

Electronically approved by prof. Ing. Pavel Tvrdík, CSc. on 6 February 2023 in Prague.

Master's thesis

PARALLEL FACTORIZATION ON GPU USING CUDA AND METAL API

Bc. Jan-Jakub Fleišer

Faculty of Information Technology
Department of Computer Systems
Supervisor: doc. Ing. Ivan Šimeček, Ph.D.
January 8, 2024

Czech Technical University in Prague
Faculty of Information Technology
© 2024 Bc. Jan-Jakub Fleišer. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Fleišer Jan-Jakub. *Parallel factorization on GPU using CUDA and Metal API* . Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgments ix

Declaration x

Abstract xi

Acronyms xii

Introduction 1

1 Algorithms 3
- 1.1 Lenstra Elliptic curve factorization . . . 3
  - 1.1.1 Elliptic curves . . . 3
  - 1.1.2 Elliptic curve groups . . . 4
  - 1.1.3 Scalar multiplication on Elliptic curves . . . 5
  - 1.1.4 Lenstra's elliptic curve algorithm . . . 6
- 1.2 Pollard's Rho . . . 7
- 1.3 Arbitrary precision integers . . . 7
- 1.4 The GNU Multiple-Precision Arithmetic Library and LibTomMath . . . 8
- 1.5 Existing implementations and approaches . . . 9
  - 1.5.1 Revisiting ECM on GPUs . . . 9
  - 1.5.2 PARI and GMP-ECM . . . 9
  - 1.5.3 Symbolic computing in Python . . . 10

2 Parallelization 11
- 2.1 OpenMP Library . . . 11
- 2.2 General Purpose computing on the GPU . . . 12
  - 2.2.1 Arbitrary precision arithmetic on GPU . . . 12
  - 2.2.2 GPU Flow-Control and divergence . . . 12
- 2.3 NVIDIA CUDA . . . 13
  - 2.3.1 CUDA Kernels . . . 13
  - 2.3.2 Threads, blocks, and grids . . . 13
  - 2.3.3 Device and Host memory transfers . . . 15
  - 2.3.4 CUDA Memory Hierarchy . . . 15
  - 2.3.5 CUDA Streams . . . 15
- 2.4 Apple Metal . . . 16
  - 2.4.1 Metal Shading Language . . . 16
  - 2.4.2 Metal-cpp . . . 16
  - 2.4.3 Metal Data Types . . . 17
  - 2.4.4 Address spaces . . . 17
  - 2.4.5 Thread grids and Thread groups . . . 17
  - 2.4.6 Metal work submission . . . 18
  - 2.4.7 Apple GPUs . . . 20

3 Sequential and OpenMP implementations 21
3.1 Sequential implementations . . . 21
3.1.1 Pollard's Rho . . . 21
3.1.2 Lenstra's Factorization . . . 22
3.2 OpenMP based implementation . . . 22
3.2.1 Lenstra's Factorization . . . 23
3.2.2 Pollard's Rho . . . 24
3.3 CPU Versions comparison . . . 25
3.3.1 Comparison . . . 25
3.4 Measurement summary . . . 26
3.5 Profiling the CPU implementation . . . 26
3.6 Summary . . . 28

4 Multi-precision integer arithmetic on Apple Metal 31
4.1 Metal Arbitrary Precision library . . . 31
4.2 Storing arbitrary precision integers . . . 32
4.3 Memory allocation limitations in Apple Metal and impact on implementation . . 32
4.4 Fixed-size integer representation . . . 34
4.5 Metal Arbitrary Precision library function conventions . . . 34
4.6 Metal Arbitrary Precision library structure . . . 35
4.7 Supported functions . . . 36
4.8 Implementation differences between CPU and GPU function versions . . . 38
4.9 Working with a large number of arbitrary precision integers on the GPU . . . 38
4.10 Encoding and using arbitrary precision integers in GPU shaders . . . 39
4.10.1 Random number generation on Apple Metal . . . 41
4.11 Summary . . . 41

5 Paralellization using Metal API 43
5.1 High-level principle of the Metal implementations . . . 43
5.2 Achieving scalability for Apple Metal . . . 43
5.3 Code layout and usage of algorithms on Metal . . . 44
5.4 Lenstra implementation on Metal . . . 45
5.5 Pollard's Rho implementation on Metal . . . 48
5.6 Performance optimizations . . . 49
5.6.1 Kernel modularity and interface . . . 50
5.6.2 Kernel complexity . . . 50
5.6.3 Memory layout . . . 51
5.6.4 Utilizing 16-bit GPU registers . . . 51
5.6.5 Utilizing fixed-size large integers . . . 52
5.6.6 Modular reductions to reduce memory bottlenecks . . . 52
5.6.7 Version and parameter performance impact . . . 54
5.7 Summary . . . 55

6 Paralellization using CUDA 57
6.1 Arbitrary integer precision arithmetic on CUDA . . . 57
6.2 Storing arbitrary precision integers . . . 57
6.3 Extending the Metal Arbitrary Precision library . . . 58
6.4 High-level principle of the CUDA implementations . . . 58
6.5 Lenstra implementation on CUDA . . . 59
6.6 Pollard's Rho implementation on CUDA . . . 60
6.7 Performance optimizations . . . 60
6.7.1 Modular reductions to reduce memory bottlenecks in ECM . . . 60
6.7.2 Utilizing fixed-size large integers . . . 61

6.7.3 Using shared memory for N . . . 61
6.7.4 Using 16-bit word size . . . 62
6.7.5 Version and parameter performance impact . . . 62
6.8 Summary . . . 63

7 Results analysis 65
7.1 Considerations for measurements . . . 65
7.2 MAP Library and GMP comparison . . . 66
7.3 CUDA and Metal implementation comparison . . . 66
7.4 CPU and GPU implementation comparison . . . 69
7.5 SymPy, PARI and GMP-ECM implementation . . . 71

8 Conclusion 77
8.1 Outcomes . . . 77
8.2 Shortcomings . . . 78

A Selected algorithms for multi-precision arithmetic 79
A.1 Considerations for the selected algorithms . . . 79
A.2 Addition and subtraction . . . 79
A.3 Greatest common divisor and extended greatest common divisor . . . 82
A.4 Multiplication and division by two . . . 82

Contents of enclosed CD 89

List of Figures

1.1 Point addition and doubling on elliptic curves as presented in *Guide to elliptic curve cryptography* [4] . . . 4

2.1 CUDA Memory Hierarchy [31] . . . 14
2.2 CUDA Grid execution [31] . . . 14
2.3 Apple Metal threadgroup split into two 16 thread SIMD-groups [34] . . . 18
2.4 Apple Metal thread grid and threadgroups[34] . . . 18
2.5 Apple Metal Compute submission on M1. The diagram shows the workflow for submitting work to the GPU by committing or enqueuing (to be committed later) the commands. [42] . . . 19
2.6 Apple M1 GPU cache [42] . . . 20

3.1 Measured runtimes per algorithm and version with eight threads (ARM64). . . . 27
3.2 Time until solution for `v7` version of Pollard's Rho for different thread counts (ARM64 M2). . . . 28
3.3 Heaviest stack trace for sequential Pollard's Rho implementation. . . . 28
3.4 Heaviest stack trace for sequential Lenstra implementation. . . . 29

5.1 The layout of a factorization class for Metal. The diagram shows the distribution of individual instances across CPU threads and their submission to the GPU device given N CPU threads, each executing eight parallel instances. . . . 45
5.2 Profiling tool output for fixed size 16-bit ETE ECM variant. . . . 50
5.3 Runtime shader costs for 16-bit fixed size point doubling in ETE ECM. . . . 51
5.4 Aggregated meantime for individual Metal variants. Capped at upper time limit. 53
5.5 Percentage impact of additional modular reductions on the meantime. . . . 54

6.1 Mean time percentage change when additional modular reduction was applied. . 61
6.2 Aggregated meantime for individual CUDA variants. Capped at upper time limit. 62
6.3 Mean time for various block thread sizes on 16-bit fixed ECM-ETE . . . 63

7.1 Comparing time required for sequential GMP library operations vs CUDA MAP library parallel operations. . . . 67
7.2 Mean runtime and single instance time for ETE EC point doubling kernels over specific grid sizes. Note that the Metal variant is limited to 4,096 due to texture width limitations. . . . 69
7.3 Mean execution time for CUDA and Metal compared to grid size (fixed-size variants). Shows particular configuration on 80-bit (ECM) numbers and 73-bit (Pollard's Rho) numbers. . . . 70
7.4 Growing maximum RSS for Metal variants on 80-bit composite input. . . . 70
7.5 Shows log-scaled runtime comparison across implemented variants. The first plot shows CPU variants together with CUDA ECM. The second plot shows all GPU variants together. . . . 72
7.6 Runtime comparison of selected, better-performing implementations . . . 74
7.7 Normalized mean runtime for growing composite size. . . . 74

7.8 Log scaled mean runtime for growing composite size 75

List of Tables

7.1 Measured mean maximum RSS for 80-bit composite input. 71
7.2 Mean runtime across all solutions. 73

List of code listings

1.1 Multi-Precision structure from *Multi–Precision Math* [11] 8
2.1 Simple vector element addition using CUDA. [31] 13
2.2 Addition computational kernel in Apple Metal. [39] 16
3.1 Pollards Rho factorization function. 21
3.2 ECM factorization function. 22
3.3 OpenMP based factorization function. 23
3.4 Optimization flags used for OpenMP variants. 25
4.1 MAP Library dynamic sized integer structure. 32
4.2 Prohibited casting in MSL. 33
4.3 Redefined MAP library structure for fixed-size integers. 34
4.4 MAP Library GPU multiplication signature. 35
4.5 MAP Library integer holder class interface. 39
4.6 Example Metal computational kernel MAP library argument passing. 40
4.7 Example CUDA kernel MAP library argument passing. 40
5.1 High-level Metal factorization class creation and usage. 45
5.2 Pseudo-code of the implemented ECM factorization logic. 47
5.3 Pseudo-code of the implemented Pollard's Rho factorization logic. 48
6.1 Comparison of MAP library CUDA and Metal variant function declarations. . . 58
6.2 CUDA Factorization class creation and usage. 59
6.3 Compilation flags used for CUDA variants. 60

List of Algorithms

1 Right-to-left binary method for point multiplication 6
2 Pollard's Rho 7
3 Low-level addition 80
4 Low-level subtraction 81

5 Binary GCD 82
6 Binary Extended GCD Algorithm 83
7 Multiplication by two 84
8 Division by two 84

I wish to express my thanks and gratitude to doc. Ing. Ivan Šimeček, Ph.D., for the provided guidance and insight. In addition, I wish to voice my gratefulness for the support I received from my family during the time of writing of this thesis.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis. I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as a school work under the provisions of Article 60 (1) of the Act.

In Prague on January 8, 2024 ....

Abstract

This thesis attempts to enable factorization using Pollard's Rho and Lesntras Elliptic curve factorization algorithms on the GPU. It goes through initial sequential CPU implementation, its adoption to a multi-threaded solution using OpenMP, and GPU-based CUDA and Apple Metal API implementations. A new multi-platform arbitrary-precision integer arithmetic library was created for Metal and CUDA to support the end goal of arbitrary precision factorization on the GPU. The thesis evaluates the performance differences across the implemented solutions and the differences between CPU, CUDA, and Metal variants. It also provides a comparison with existing noteworthy solutions.

Keywords Pollard's Rho, ECM, Factorization, Parallel factorization, arbitrary-precision arithmetic, multi-precision arithmetic, GPGPU, Apple Metal, CUDA, OpenMP, GMP

Abstrakt

Tato práce se snaží umožnit faktorizaci pomocí Pollardova Rho a Lesntrova algoritmu pro faktorizaci pomocí eliptických křivek na grafických procesorech s libovolnou přesností. Práce popisuje vytvořené implementace od počáteční sekvenční verze, přes její adaptaci na vícevláknové řešení pomocí OpenMP, a nakonec po implementace pro GPU s využitím CUDA a Apple Metal API. Pro dosažení faktorizace s libovolnou přesností na GPU je vytvořena nová multiplatformní knihovna pro aritmetiku celých čísel pro Metal a CUDA API. Práce zhodnocuje a komentuje naměřené výkonnostní rozdíly mezi implementovanými řešeními a rozdíly mezi variantami pro CPU, CUDA a Metal API. Práce poskytuje také srovnání s existujícími významnými řešeními ve světě celočíselné faktorizace.

Klíčová slova Pollard's Rho, ECM, Faktorizace, paralelní factorization, výpočty s libovolnou přesností, GPGPU, Apple Metal, CUDA, OpenMP, GMP

Acronyms

API	Application Programming Interface
CPU	Central processing unit
CUDA	Compute Unified Device Architecture
ECM	Elliptic-curve factorization method
ETE	Extended Twisted Edwards
GPGPU	General-purpose computing on the GPU
GPU	Graphics processing unit

Introduction

This thesis explores the application of general-purpose computing on the GPU in Apple Metal and NVIDIA CUDA technologies for arbitrary precision integer factorization. One possible application of integer factorization is finding prime factors of large composite numbers, as some cryptosystems, such as RSA, rely on the difficulty of the factorization problem to remain secure. [1] This thesis focuses on two algorithms, Pollard's Rho and Lenstras Elliptic curve factorization also known as ECM, it walks through selected existing solutions and introduces numerous implementations in various forms, starting with sequential implementations, those are then extended to a parallelized multi-threaded solution, finally, the main focus of this thesis, new implementations are considered for Apple Metal and CUDA APIs, which attempt to leverage GPUs for majority of the required computations. To achieve this, a new arbitrary-precision library is introduced, explicitly targeting Apple Metal and CUDA, enabling various use cases. Finally, the new implementations are compared to several noteworthy existing solutions in the factorization domain.

Chapter 1
Algorithms

This thesis focuses on two algorithms for integer factorization. One is Pollard's Rho algorithm, and the second is Lenstra's Elliptic Curve factorization. In this chapter, the algorithms used are briefly introduced and described. The chapter also gives an introduction to existing noteworthy implementations of these algorithms, as well as some more sophisticated factorization solutions. Lastly, the chapter discusses the problematic of enabling arbitrary-precision integer arithmetic.

1.1 Lenstra Elliptic curve factorization

This section describes H. W. Lenstra's algorithm for factorization of positive numbers based on Elliptic curves as described in *Factoring integers with elliptic curves* [2]. The algorithm is also frequently referred to as the elliptic-curve factorization method or shortly ECM, and the names will be used interchangeably throughout the text. Naturally, there is a multitude of algorithms for factorization, and ECM appears widely in existing solutions or libraries, making it an interesting choice. While ECM is relevant amongst factorization algorithms, different methods, such as the numbers field sieve, can generally be considered more efficient. [3] Despite this, ECM is widely used for factorization cases with composite integers for which no information about the prime factors' sizes is available. Additionally, ECM can be found within a *cofactoring* step of the number field sieve. [3] First, some of the core concepts, such as the description of elliptic curves and the elliptic curve groups, are discussed, followed by the description of the algorithm.

1.1.1 Elliptic curves

An elliptic curve E in the Weierstrass form over a field K is given as $y^2 = x^3 + ax + b$ together with a point at infinity O where $4a^3 + 27b^2 \neq 0$. [4] Or as a set:

$$\{(x, y) \in \mathbb{R}^2 : y^2 = x^3 + ax + b\} \cup \{O\}$$

While various curve forms exist, such as Montgomery, Edwards, or twisted Edwards, the implementations in this thesis are limited to the Weierstrass and twisted Edwards forms. The usage of different forms of elliptic curves can be seen through various existing implementations of the ECM algorithm. To provide some examples, the `GMP-ECM` implementation of ECM relies on Montgomery curves, as does the *SymPy* library, which take the form of $by^2 = x^3 + ax^2 + x$. [3] Another prominent form is Edwards curves, which take the form of $x^2 + y^2 = 1 + dx^2y^2$ with $d(d - 1) \neq 0$. [3] One notable implementation utilizing Edwards curves is EECM-MPFQ appearing in [5]. Twisted Edwards curves as introduced in *Twisted Edwards Curves* [6] generalize

Edwards curves with the form of $ax^2+y^2 = 1+dx^2y^2$ with $ad(a-d) \neq 0$. This form gained some prominence due to the favorable properties of the arithmetic in the elliptic curve group. [3] An Edwards curve is a twisted Edwards curve with $a = -1$. [6] Specifically, the Twisted Edwards form where $a = -1$ is particularly interesting as it offers fast scalar multiplication. [3] Twisted Edwards curves with $a = -1$ form can be seen in the `ecmongpu` solution which will be discussed in later sections.

1.1.2 Elliptic curve groups

An abelian group can be formed over the set of points $E(K)$ with O as identity and an operation of point *addition* where individual points $P, Q \in E(K)$ on the elliptic curve are added $R = P+Q$. [4] The geometric visualization of the addition and doubling of points on elliptic curves can be observed in figure 1.1. This operation can be seen as *addition* when P and Q are distinct or *doubling* when $P = Q$.

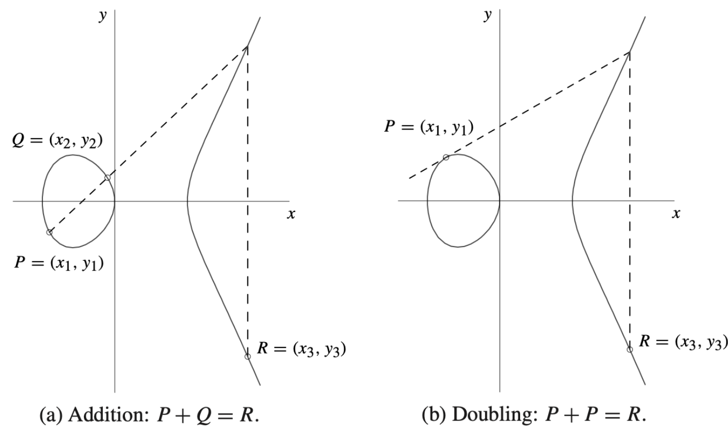

Figure 1.1 Point addition and doubling on elliptic curves as presented in *Guide to elliptic curve cryptography* [4]

Considering elliptic curves of the Weierstrass form, an abelian group $(G, +)$ with a set of points G on an elliptic curve E over a field K in affine coordinates can be defined with $+$ operation. [4] The $+$ binary operation over $P = (x_1, y_1)$ and $Q = (x_2, y_2)$ points on the curve can be defined as $R = (x_3, y_3) = P + Q$ with:

$$(x_3, y_3) = (\lambda^2 - x_1 - x_2, \lambda(x_1 - x_3) - y_1)$$

and λ being defined separately for $P \neq Q$ and $P = Q$. [7] For the first case, where $P \neq Q$, λ is defined as:

$$\lambda = (y_2 - y_1) * (x_2 - x_1)^{-1}$$

and for the case of $P = Q$, λ is:

$$\lambda = (3x^2 + a) * (2y_1)^{-1}$$

In order to compute the inverses shown in computations of λ, the extended Euclidean algorithm can be used. The addition and doubling, as defined above, require computing inversions, which is a computationally expansive operation. In such cases, it can be favorable for implementation performance to change how the points are represented, such as by utilizing projective coordinates. [4] In order to compute scalar multiplication kP faster, a different point representations as described in *Twisted Edwards curves revisited* [8] can be used. First, an auxiliary coordinate

$t = xy$ to point (x, y) on the $ax^2 + y^2 = 1 + dx^2y^2$ curve is introduced to represent the point in extended affine coordinates. Then, to pass to a projective representation, the mapping $(x, y, t) \mapsto (x : y : t : 1)$ is used. This representation is referred to as *extended twisted Edwards coordinates*, with $(0 : 1 : 0 : 1)$ as the identity element. [8] Interestingly, this representation can be seen in the *ecmongpu* GPU implementation for CUDA in *Revisiting ECM on GPUs*. [9]

Coming back to the elliptic curve forms, specifically the twisted Edwards curves, the *affine* addition operation + on points P and Q on an elliptic curve E over a field K with odd characteristics can be defined as:

$$R = (x_3, y_3) = P + Q = \left(\frac{x_1y_2 + y_1x^2}{1 + dx_1y_1x_2y_2}, \frac{y_1y_2 - ax_1x_2}{1 - dx_1y_1x_2y_2} \right)$$

with $(0, 1)$ neutral element. [6] This formula is also applicable for point doubling when $P = Q$. The formula is complete (has no exceptional cases) if d is not square in K and a is square in K. [6] For twisted Edwards curves with $a = -1$ the addition operation $P + Q = (X_1 : Y_1 : T_1 : Z_1) + (X_2 : Y_2 : T_2 : Z_2) = (X_3 : Y_3 : T_3 : Z_3) = R$ using extended twisted Edwards coordinates is defined by *Twisted Edwards curves revisited* [8] as:

$$\begin{aligned}
A &= (Y_1 - X_1) * (Y_2 + X_2) & B &= (Y_1 + X_1) * (Y_2 - X_2) \\
C &= (2Z_1 * T_2) & D &= 2T_1 * Z_2 \\
E &= D + C & F &= B - A \\
G &= B + A & H &= D - C \\
X_3 &= E * F & Y_3 &= G * H \\
T_3 &= E * H & Z_3 &= F * G
\end{aligned}$$

While doubling for extended twisted Edwards coordinates is defined as:

$$\begin{aligned}
A &= X_1^2 & B &= Y_1^2 & C &= 2Z_1^2 & D &= aA \\
E &= (X_1 + Y_1)^2 - A - B & G &= D + B & F &= G - C & H &= D - B \\
X_3 &= E * F & Y_3 &= G * H & T_3 &= E * H & Z_3 &= F * G
\end{aligned}$$

Further, the mentioned article discusses further advanced techniques, such as mixing different coordinate types during the computation and parallelization for computational speedup. These approaches are, however, not utilized in this thesis.

1.1.3 Scalar multiplication on Elliptic curves

One of the critical operations for Lenstra's ECM algorithm and elliptic-curve cryptography is the scalar multiplication or point multiplication, kP with k integer and P a point on an elliptic curve E over a field K. [4] One of the methods for computing point multiplication, and the one used in this thesis is the right-to-left binary method as shown in algorithm 1. Given k in binary representation, each bit is processed, and in each iteration, the point is added and doubled for bits set to 1, or only doubled for 0 bits. This operation represents a basic form of repeated square-and-multiply method for addition. [4] While additional methods exist, such as *Window*, *Montgomery*, or *Fixed point* and can even provide better run time [4], those are not making an appearance in this thesis.

Algorithm 1 Right-to-left binary method for point multiplication

Require: $k = (k_t, ..., k_1, k_0)_2$, $P \in E(K_q)$.
Ensure: kP.
1: $Q \leftarrow \mathcal{O}$.
2: **for** $i = 0$ up to k_t **do**
3:     **if** $k_i = 1$ **then**
4:         $Q \leftarrow Q + P$.
5:     **end if**
6:     $P \leftarrow 2P$.
7: **end for**
8: **return** Q.

1.1.4 Lenstra's elliptic curve algorithm

Lenstra's ECM is a general-purpose factoring algorithm on elliptic curves defined modulo factored composite number. The algorithm operates over a group of points on the elliptic curve. [9] The original formulation of the ECM algorithm as introduced by H. W. Lenstra Jr. in *Factoring integers with elliptic curves* [2] used Weierstrass curves. Today, the usage is more varied, and different curve forms are common, such as Montgomery, Edwards curves, or twisted Edwards curves, as mentioned in previous sections. [3] The choice of elliptic curves plays a prominent role in the algorithm and is a widely studied subject.

The ability of the ECM algorithm to find a factor depends on the smoothness of the elliptic curve order, with different random elliptic curves likely resulting in different group orders. Thus, running multiple ECM instances with random elliptic curves, with random orders, in parallel over the same factored number increases the probability of finding a factor. [9] The ECM algorithm is considered "embarrassingly parallel" with no dependence between attempted trials. For a number of trials much greater than the number of available processors P, linear speedup is possible. However, achieving this in practice may be difficult due to memory constraints. [1] The ECM algorithm consists of two stages, with the second one being optional. [9] Only the first stage is considered in this thesis, and it goes as follows: a random elliptic curve E over a field $\mathbb{Z}_n$ is selected with n being the composite number with factor p, as well as a random point P on the curve E. The value kP is computed, with k being a large scalar. The sought-after result of the kP multiplication is the identity element O on the elliptic curve modulo p but not modulo n, with p being unknown, all computation on the curve are being performed over $\mathbb{Z}_n$. With k as a multiple of curve order kP is equal to $O = (0 : 1 : 0)$ modulo p but not modulo n. In such case, the x and z coordinated are multiples of p, and thus, computing $gcd(z, n)$ should reveal the sought-after factor of n. As for the value of the large scalar k, it is usually picked as a product of small powers of primes ranging from 1 up to an upper boundary B, giving $k = lcm(1, 2, 3, ...B)$. [9]

A notable mention about the ECM algorithm is its similarity to Pollard's *p-1* algorithm, which, in principle, can be regarded as an attempt to find identity in a group. [1] In Pollard's algorithm we can choose k as $k = lcm(1, 2, 3, ..., K), K \leq \sqrt{n}$ and $1 < a < n$. With this, the value of $gcd(a^k - 1, n)$ can be computed. If $p - 1|k$, the value of p will be found. If the computation fails, K can be increased. The problematic aspect of the algorithm is if n does not have a factor p of where $p - 1$ would be a product of small primes to small powers. This will result in a need to raise the value of K, considerably affecting the algorithm. [7] In ECM, instead of raising to the power of k, we can compute kP of a point $P \in E(\mathbb{Z}_|)$ and if the order of the curve divides k, then $kP = O$ and a factor will be found. As was mentioned, the order can vary depending on the *randomly* selected curve. This means that if this fails, a new random curve can be picked, likely resulting in a different order. [7]

1.2 Pollard's Rho

Pollard's Rho algorithm, as originally described by John. M. Pollard in *A Monte Carlo Method For Factorization* [10] is a factorization method that relies on probabilistic ideas to find the factors of an integer N. [10] The algorithm generates a sequence of values by applying a pseudo-random function $f(x)$ to the initial x_0, that is:

$$x_{i+1} = f(x_i) \mod N$$

Where x_0 is chosen randomly. The function $f(x)$ is typically chosen to be a polynomial, such as

$$f(x) = x^2 + a$$

. Where $a \neq 0$ is a constant chosen randomly. From this sequence, the value of:

$$gcd(|x_{2i} - x_i|, N)$$

is computed for each $i = 1, 2, ...$, and if the result is nontrivial GCD, this result is the factor of N, and the computation is stopped. [1] The above explanation is provided in the pseudo-code 2 for better understanding. One approach to the p of the algorithm is to try a number of different pseudo-random sequences as generated by different polynomials f. [1] For P processors with P different parallel sequences, the speedup is $\theta(P^{1/2})$. [1]

Algorithm 2 Pollard's Rho

Require: Composite integer n
Ensure: Factor of n or -1 for failure
1: $x_i \leftarrow 2, x_j \leftarrow 2$
2: **while** $d = 1$ **do**
3: $\quad x_i \leftarrow f(x_i) \bmod n$
4: $\quad x_j \leftarrow f(f(x_j)) \bmod n$
5: $\quad d \leftarrow \gcd(|x_i - x_j|, n)$
6: **end while**
7: **if** $d = n$ **then**
8: $\quad$ **return** -1 ▷ Failed to find a factor
9: **else**
10: $\quad$ **return** d
11: **end if**

1.3 Arbitrary precision integers

This section briefly discusses the problematic of working with arbitrary precision or multi-precision integers as occasionally referred to, such as in *Multi–Precision Math* [11]. In most cases, standard computations use fixed precision integers for calculations. These use a constant amount of memory, frequently 16, 32, or 64 bits per number that needs to be stored. This, naturally, limits the size of the number that can be stored. However, some computations require working with larger numbers, which may not be possible to store within these fixed sizes. To circumvent those obstacles, it is possible to implement custom data formats with variable amounts of memory for storage. Through this thesis, those will be referred to as multi-precision or arbitrary precision integers. To enable factorization of multi-precision integers, a proper representation of the numbers needs to be defined and the necessary arithmetic operations implemented. Common representations usually store individual parts of the number, frequently referred to as limbs, as

a sequence of fixed-size integers, as does the representation chosen in this thesis, which will be discussed in detail in later chapters.

While different implementations vary in how the arbitrary-precision integers are represented, the representations need to contain the same information. This is the sign of the integer, the individual limbs from which the integer is composed, the number of such limbs, and, in the case of a variable-sized integer, the number of available, allocated limbs. To provide an example, in *Multi–Precision Math* [11] the following struct is used to represent the integers:

Code listing 1.1 Multi-Precision structure from *Multi–Precision Math* [11]

```
typedef struct {
    int used, alloc, sign;
    mp_digit *dp;
} mp_int;
```

For comparison, the widespread GMP library uses a slightly different representation. The GMP representation is more compact, as it uses a signed size variable, which stores the number of used limbs either as a positive or negative number, which also indicates the overall sign. This removes the need for a separate variable, as shown in the example above. For GMP, as was the case in the example, the limbs are accessed through a pointer to an array of limbs stored in a “little-endian” fashion. [12] With the representation settled, the focus has to shift toward algorithms that support multi-precision integer arithmetic. Since the number of algorithms to cover even basic functionality is very large, only a very limited section is shown in the Appendix. The mentioned algorithms are derived from the *Multi–Precision Math* [11] and *Handbook of Applied Cryptography* [13] sources, which describe a wide set of algorithms for multi-precision integer operations. These sources have been used to implement the MAP library mentioned in later chapters.

1.4 The GNU Multiple-Precision Arithmetic Library and LibTomMath

As briefly mentioned in the previous chapter, various formats and algorithms exist to enable arbitrary precision computations. This section presents the GMP and *LibTomMath* libraries, which provide this functionality. The libraries provide all the necessary functionality required to implement the factorization algorithms in arbitrary precision, ranging from simple arithmetic operations such as addition up to more complex algorithms such as computing modular inverse. While only the GMP library was used in the CPU-based implementations, the *LibTomMath* library served as an excellent resource for study and later implementation of a similar arbitrary-precision Metal and CUDA library. *LibTomMath* saw its first version in 2002, and at the time of writing of this thesis, it has 39 contributors on its GitHub page. [14] The library provides a variety of operations, from basic operations like addition, subtraction, division, and multiplication to more complex operations such as modular reductions or GCD. Some of the operations are supported by a variety of implemented algorithms.

The GMP library was first released in 1991 and was continually developed with a significant list of contributors, which are listed under release-version on the library’s home page [15]. GMP and *LibTomMath* are well-established libraries with years of development by a larger group of people, and it is improbable that any solution implemented in this thesis would outperform these libraries, either in functionality or optimization with the implementation of a similar library with specific requirements being only a portion of this thesis. This is why, whenever possible, the GMP library is utilized to leverage all the benefits mentioned and the years of hard work behind the library. The library has capabilities beyond multi-precision integer arithmetic (which is supported by about 150 functions in this category), providing functionality for rational and floating-point arithmetic. As was the case for *LibTomMath*, GMP provides a variety of algorithms to support

specific operations. It should be noted that a variety of algorithms for specific operations exist. To provide an example, the GMP library utilizes seven multiplication algorithms such as *Karatsuba, Basecase, Toom-3, Toom-4* and others to produce a result. The library decides which algorithm to use for the computations based on defined thresholds, aiming to provide optimal performance for the considered integer inputs. [16] This behavior is not limited to multiplication, and similar behavior can be seen in additional crucial operations such as computation of division, GCD, or extended GCD, which uses Binary GCD algorithm for small inputs, followed by Lehmer's algorithm up to a specific threshold and HGCD for inputs above. [16]

1.5 Existing implementations and approaches

This section discusses selected existing solutions, projects, and implementations of the integer factorization problem and published papers discussing the topic. Four notable existing solutions will be covered, and three of them will reappear in later chapters for comparison with the implementation created in this thesis. The mentioned solutions cover a very small subset of available implementations, adaptations, and publications. They have been chosen either because of their prevalence or closeness to the approach chosen in this thesis.

1.5.1 Revisiting ECM on GPUs

The first discussed solution appears in the *Revisiting ECM on GPUs* paper [9]. This paper uses and implements a two-stage Elliptic Curve method with $a = -1$ twisted Edwards curves for CUDA-enabled GPUs to accelerate the computation. This implementation is close to the considered approach selected for this thesis, making this implementation highly desirable for comparison. The publication presents a highly optimized two-stage Lenstra's ECM algorithm for GPUs referred to as `ecmongpu` (although the second stage can be disabled). The implementation relies on 32-bit unsigned limbs (Configuration also offers a 64-bit option during compile-time) to represent the integers. The solution relies on fixed-size multi-precision integers, with the length being defined at compile time. The benefit of this is that it allows loop-unrolling by the compiler and additional possible optimizations. [9] Further, the implementation relies on PTX code (Parallel Thread Execution - low-level parallel thread execution virtual machine and ISA) to deliver highly optimized and performant multi-precision arithmetic. Another remark is that the implementation relies on the GMP library for CPU-based computations.

1.5.2 PARI and GMP-ECM

The following discussed solutions can be accessed and interfaced through SageMath, which provides a convenient high-level interface. Sage provides functions for factorization, which utilize PARI and GMP-ECM solutions. PARI is an open-source algebra system for fast number theory computations, including factorization. PARI is available as a C library, through an interactive shell or, as mentioned earlier, through an interface with SageMath. PARI contains a sophisticated factoring engine, which includes Pollard's Rho and ECM utilizing Montgomery curves. Among those, additional methods are present, such as Square form factorization, multiple Polynomial Quadratic Sieve, and a search for pure powers. The documentation does not clearly state how, but a combination of those algorithms is utilized to acquire the factors, providing a fast, highly performant solution for factorization. [17]

GMP-ECM is a highly optimized implementation of Lenstra's Elliptic curves algorithm with two stages. The solution utilizes curves in Montgomery form, and unsurprisingly, the arithmetic in the library is supported by the GMP library. GMP-ECM also has the capability to run the first stage of the algorithm on CUDA GPUs. The GPU implementation attempts to fit the maximum possible number of curves to fully utilize the GPU. The GPU code relies on the *CGBN: CUDA*

Accelerated Multiple Precision Arithmetic library and is limited to compile-time defined fixed size of the input ranging from 256 to 32,768 bits. [18] The mentioned CGBN library provides the capability of highly-optimized multiple precision integer arithmetic in CUDA and utilizes *cooperative group of threads* that work together to represent and process operations on each big number. [19]

1.5.3 Symbolic computing in Python

Finally, an easily accessible solution is the *Symbolic computing in Python* library, or simply SymPy. [20] As the name suggests, it is a Python library focusing on symbolic mathematics. The library offers various functionality ranging from geometry, calculus, cryptography, and statistics to factorization, for which it implements both Pollard's Rho and Lenstra's Elliptic curves algorithms. The ECM in SymPy considers elliptic curves in Montgomery form, is implemented in two stages, and utilizes trial division for smaller factors. [21] While the library is written in Python and with different principles in mind over the implementation in this thesis, the fact that SymPy is a well-known and widespread library with robust implementation indicates that it could provide a good baseline. The library can be utilized sequentially, or a similar parallelism over the library functions can be created for future comparison.

Chapter 2
Parallelization

This chapter focuses on the technologies, frameworks, and libraries allowing the parallelization of the previously introduced algorithms within this thesis. The chapter starts with a brief introduction to OpenMP for CPU parallelization. Later, heavier emphasis is put on describing the principles and fundamentals of Metal and CUDA APIs for parallel GPU computations.

2.1 OpenMP Library

The OpenMP Library, managed by the OpenMP Architecture Review Board consortium, provides an API that enables shared-memory multiprocessing for a variety of languages, platforms, and operating systems. [22] It utilizes a fork-join parallel execution model, with tasks being executed implicitly or explicitly as given by specified OpenMP constructs and directives. [23] The library provides various constructs for parallel computations. In this thesis, the key utilized constructs and directives are *parallel for*, *tasks*, and *parallel regions*. For each of those, a brief overview is given below:

- `parallel` construct - when a parallel construct is encountered, it creates a number of threads that execute the code defined within the region. [23]
- `parallel for` work sharing-loop construct - specifies that an iteration of one or more loops should be executed by threads in parallel. [23]
- `task` construct - A task generating construct. When encountered, an explicit task is created from the associated block of code, and any thread may be assigned the task to execute. [23] A related construct is `taskgroup`, which specifies that completion of all child tasks and their descendants should be awaited. [23]

In addition to these constructs, OpenMP offers easy-to-use critical sections and atomic operations. This includes operations such as `atomic read` and `atomic write`, which state that a specified storage location should be written to or read atomically. [23] While other atomic constructs are available within the library, they are not utilized within this thesis. The last mentioned construct is `critical`, the usage of which restricts the execution within the associated block of code to a single thread at a time. [23] The used constructs within this thesis keep to a small subset of capabilities and functionality offered by the OpenMP library.

2.2 General Purpose computing on the GPU

General-purpose computing on the GPU (GPGPU) refers to the practice of utilizing the hardware capabilities provided by GPUs for solving general computational problems. While using GPU hardware specificity may result in performance benefits for certain applications, it requires adjusting the code to fit the GPU computational environment better. It should also be noted that not all problems necessarily benefit from this adaptation. They may encounter various bottlenecks or do not have to be easily adjustable for GPUs. [24]

In order to adapt and write code for GPGPU, an API that will enable access to the GPU is required. This thesis is limited to CUDA and Apple Metal, which, when considering newer hardware, support disjunctive hardware. CUDA is a widespread and heavily utilized API for GPGPU, appearing in various applications. One such example could be the TensorFlow library [25], or, closer to the topic of this thesis, the GMP-ECM (optional stage 1) or *ecmongpu* ECM implementations. In contrast, Apple Metal seems to be significantly less common, both generally from a GPGPU perspective or from a factorization perspective, with no published ECM implementation on Metal. This may not be entirely surprising as Apple Metal specifically targets GPUs available mainly in Apple devices such as Mac computers or mobile and tablet devices [26], which will likely not provide the same scalability as CUDA-enabled datacenters with GPUs, making it a less appealing technology to target. Even with that, the power efficiency and performance of newer Apple M1 and M2 chips, alongside their unified memory, could make Metal more viable as a GPGPU API in the future, at least for a specific portion of applications.

2.2.1 Arbitrary precision arithmetic on GPU

The GMP and LibTomMath implementations are intended to run on the CPU and cannot be easily used for GPGPU computations. However, several other libraries exist for implementing arbitrary precision arithmetic on GPUs using CUDA, such as the previously mentioned cooperative *CUDA Accelerated Multiple Precision Arithmetic* library [19] supporting integer arithmetic and promising significant speedup over GMP, or *CAMPARY - Cuda Multiple Precision Arithmetic Library* [19] and *gpuprec - Extended-Precision Libraries on GPUs* library [27] which supports mostly basic operations for real numbers. As well as implementations that were implemented within theses focused on multiple-precision GPU computations, as is the case for Langer [28] and Petrouš [29], which supports basic integer arithmetic but seem to lack support for more complex operations such as GCD or modular reduction, those implementations seem to be intended to parallelize the arithmetic operations. As a final comment, the *ecmongpu* library seems to rely on its own implementation of multi-precision arithmetic. At the time of writing of this thesis, no similar widespread implementation or library has been available for Apple Metal.

2.2.2 GPU Flow-Control and divergence

Given the highly parallel architecture of GPU hardware, writing code targeting GPUs requires caution for flow-control of the code as diverging code execution paths, such as which occur during branching of the code, result in divergence and may significantly hurt performance. [30] This occurs when a SIMD (Single instruction, multiple data) group executes code where some threads take one path while others take a different one, with both CUDA and Metal applications being affected, with the impact being heavily dependent on the specific hardware. As a note, NVIDIA uses the SIMT (Single instruction, multiple threads) terminology as a less restrictive definition, with the distinction of SIMT instructions specifying the execution and branching behavior of a single thread. [31] The way divergence is handled by the hardware and the performance problems it may introduce is unsurprisingly architecture-dependent. One example of this is the NVIDIA Volta architecture, which supports independent thread scheduling. [32] Volta independent thread

scheduling enables interleaved execution of statements from divergent branches. [33]

The divergence phenomenon is also present in Apple Metal applications, where so-called SIMD-groups execute the same code, where each branch needs to be executed by all threads, resulting in execution time being a sum of both branches. [34] The inefficiency here is that each core within the group executes all paths for each branch, resulting in reduced performance as all branches need to be executed. [30] As was mentioned, one of the sources is branching in the code. GPU code should minimize the amount of branching for better performance. [30]

2.3 NVIDIA CUDA

This section discusses the NVIDIA CUDA API's specificity, features, and general principles. The section starts with how computations are organized, scheduled, and later executed on the GPU using CUDA.

2.3.1 CUDA Kernels

CUDA allows the defining of C++ functions called kernels. Kernels are defined using `__global__` specifier and called with a specific *execution configuration* syntax that defines how and by how many GPU threads the function should be executed. [31] A simple example of a kernel, as provided by the CUDA programming guide by NVIDIA, is shown below. [31] In the example, each thread adds elements from A and B and stores the results in C.

Code listing 2.1 Simple vector element addition using CUDA. [31]

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
    ...
}
```

2.3.2 Threads, blocks, and grids

A notable difference between CPU and GPU threads is that GPU threads are meant to be executed in hundreds or thousands in parallel, compensating for the slower single-thread performance and allowing for greater throughput over CPU threads. [31] In CUDA, threads are organized into one, two, or three-dimensional blocks. These blocks form either one, two, or three-dimensional compositions referred to as a grid. The hierarchy of threads, blocks, and grids can be seen in the 2.1 figure. Figure 2.2 provides a more visual illustration of both the structure and *execution* of CUDA grids. Additionally, starting with Compute capability 9.0, CUDA allows usage of Thread Block Clusters, which guarantees that blocks within the cluster are co-scheduled on the GPU Processing Cluster. This concept is not leveraged in this thesis but is mentioned to provide context for the figures used. [31]

NVIDIA GPUs have a collection of Streaming Multiprocessors. When a CUDA kernel is invoked, individual blocks get distributed amongst the multiprocessors with free capacity, with

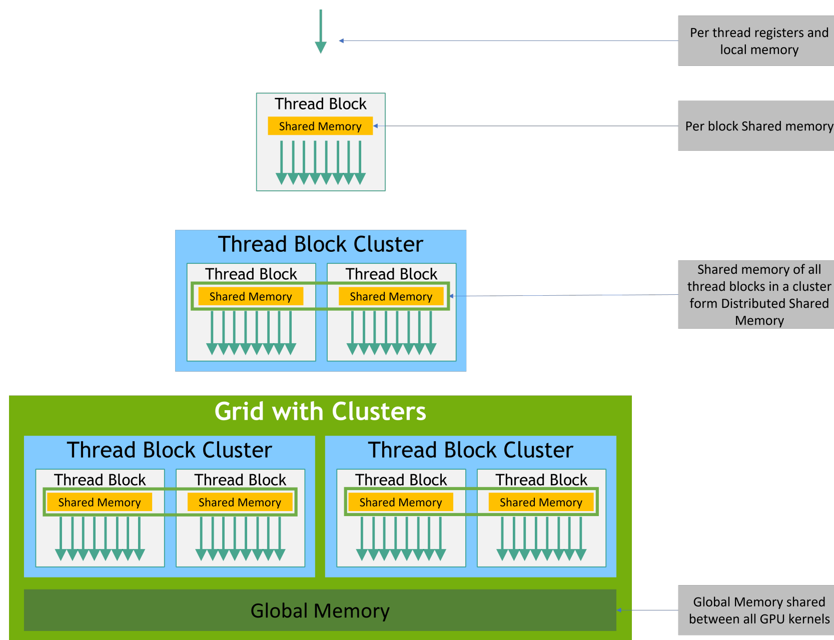

Figure 2.1 CUDA Memory Hierarchy [31]

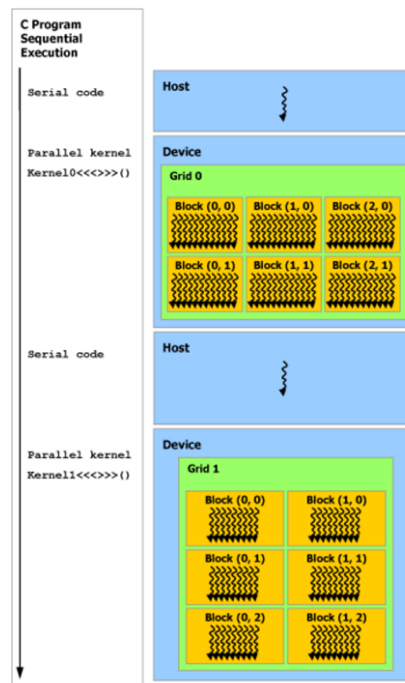

Figure 2.2 CUDA Grid execution [31]

multiprocessors being built to execute large quantities of threads concurrently. [31] The threads within a thread block are executed in parallel on one of the multiprocessors, with multiple threads being able to execute in parallel on one multiprocessor. [31] Multiprocessors create, schedule, manage, and execute threads in groups of 32, referred to as warps. Within a warp, threads start

in the same position but maintain a separate set of registers and instruction counters, which allows them to branch independently on each other. [31] Multiprocessors split thread blocks into warps of consecutive thread IDs, which get scheduled and executed. The execution is performed one instruction at a time. If threads diverge and require different instructions to be executed, each branch is executed within a warp, with threads that are on a different path being disabled. [31] Before NVIDIA Volta architecture, a single program counter was used across a warp, with masks for active threads. This could cause issues with deadlocks as divergent regions could not exchange signals or data. [31] With Volta, Independent Thread Scheduling is introduced, allowing full concurrency between threads and enabling better use of execution resources.

2.3.3 Device and Host memory transfers

The CUDA programming model assumes that the CUDA computation happens on a device that is physically separate from the host running the executing program. [31] An additional assumption is that the host and the device have separate memory, and each is referred to as host and device memory. [31] For the host and device to work on the same data, the data must be copied from host to device or from device to host. This is a potentially costly operation associated with a performance penalty. Therefore, minimizing data transfers is beneficial for performance. [35]

The CUDA programming model has a concept of unified memory. It defines a managed memory space that appears coherent to all processors with a common address space. The utilization of this mechanism allows for avoiding explicitly copying memory and greatly simplifies development. [31] In the background, the data is still copied between the host and device memory, but the code is simpler and more maintainable. [31]

2.3.4 CUDA Memory Hierarchy

A crucial concept for performance is the CUDA memory hierarchy. Data may be stored in various distinct memory spaces with varying impact on performance. The first such space is the Global memory, which is a globally accessible persistent memory located in device memory. It has high latency, low bandwidth, and is cached. Local memory is only accessible to specific threads and is automatically used in specific cases, such as arrays not accessed with constant quantities and large structures or arrays taking up significant space, or any variable if the associated kernel needs more registers than are available (*register spilling* into higher latency memory). Local memory resides in device memory and suffers from the same latency and bandwidth issues as global memory, and it is cached, too. Next is the shared memory, which offers a much higher bandwidth and lower latency over local and global memory, can be accessed by multiple threads within a thread block, and lacks cache. Constant memory resides in device memory and is cached with a constant cache. Texture memory resides in device memory and is cached with texture cache. Register are thread-specific, fast on-chip memory matching the lifespan of a warp. Memory usage also has an impact on how many blocks and warps for a specific kernel can reside on a single multiprocessor. This depends on the amount of register and shared memory the kernel needs and the amount available on the multiprocessor. [31] The memory hierarchy is further illustrated in figure 2.1.

2.3.5 CUDA Streams

The last CUDA concept that will be discussed is that of CUDA Streams. Streams allow concurrent execution of CUDA operations, which include kernel but also memory copies between device and host. Stream is a sequence of in-order operations that execute on the GPU. The benefit of

streams is that they permit running various CUDA operations concurrently and in an interleaved fashion. Without explicit specification, CUDA uses a Default stream for all operations [36]

2.4 Apple Metal

This section introduces the Apple Metal API and some of its features. The Metal API is a framework that provides applications with direct access to the host's graphics processing unit device. Metal provides low-level hardware access, efficient memory management, and a flexible compute language, which enables developers to create optimized applications that can run on a wide range of devices. [37] Note that the hardware supported by Metal is quite varied, ranging from mobile devices to older Mac devices with AMD or NVIDIA GPUs or newer devices with Apple Silicon chips. This thesis targets a specific set of newer devices and Metal versions for the computations. This will be specified in greater detail in the following sections.

2.4.1 Metal Shading Language

The Metal Shading Language is used to define work to be scheduled using Metal. The Metal Shading Language, or simply MSL, is a C++14-based language with additional extensions and restrictions, which allows writing graphic and data-parallel compute code. To give a simple example, among the restrictions is a lack of `goto` statements and `dynamic_cast` operator. The Metal Shading language works with the Metal framework, which handles the execution and compilation of Metal programs. Metal leverages Clang and LLVM, allowing it to deliver optimized GPU code. Metal has its own standard library, which is used instead of the C++ standard library. In Metal, for historical reasons, the code running on the GPU is referred to as a shader. For the Metal API, those are specified in the Metal Shading Language. [38] The same example of individual array element addition as was shown for CUDA is given below in MSL code as provided by the official Metal documentation. [39]

Code listing 2.2 Addition computational kernel in Apple Metal. [39]

```
kernel void add_arrays(device const float* inA,
                       device const float* inB,
                       device float* result,
                       uint index [[thread_position_in_grid]])
{
    // Instead of a for-loop, a collection of threads, each of which
    // calls this function is used.
    result[index] = inA[index] + inB[index];
}
```

In the example, a public GPU function is declared. The function has a void return type and is of a kernel type, making it a computational function or, alternatively, a compute kernel. [39] Compute functions are functions performing computations in parallel using a grid of threads. [39] Note the similarity with the kernel example provided in the CUDA section. One noteworthy difference is that in Metal, the thread index, or ID, is an explicit parameter, while in CUDA, it is not.

2.4.2 Metal-cpp

`Metal-cpp` is an interface enabling development for Metal in C++. This is an interface with direct mapping to Objective-C elements and provides no measurable overhead for the application [40]. `Metal-cpp` was used for the development of the Metal implementation in this thesis.

2.4.3 Metal Data Types

The Metal Shading Language supports a number of data types, with the most relevant for this thesis being signed and unsigned integers. MSL supports 8-bit, 16-bit, 32-bit, and 64-bit signed two's complement and unsigned integers. [38] Larger types are not natively supported. The chosen data type can significantly affect shader performance. Apple GPUs are optimized for 16-bit data types, with larger data types consuming more registers (potentially causing register spillover). Using 16-bit data types can increase occupancy, allowing more GPU threads to run at the same time. As an additional benefit, 16-bit data types, in most cases, utilize faster 16-bit arithmetic instructions. [41] Registers are allocated to kernels in register blocks. This means that when attempting to reduce the usage, the reduction needs to be block-sized to see improvements. Large arrays or structures can consume a large number of registers. [42]

2.4.4 Address spaces

As could be seen in the function example in the previous section, the pointer parameters are preceded by the `device` keyword. This keyword refers to the address space, which has to be specified for all pointers and references in MSL. The Metal Shading language contains multiple disjoint address spaces for memory. [39] The address space attribute declares in which region of the memory the objects are allocated. [39] The following list shows a selection of address spaces as defined in the MSL Specification [38]:

- `device` - Read-write memory allocated from the device memory pool.
- `constant` - Read-only device memory, as device memory is allocated from the device memory pool.
- `thread` - Per-thread memory, not visible to other threads. Variables declared in kernel functions are allocated in this address space.
- `threadgroup` - Shared memory between concurrently running threads in a *thread group*. On most devices, *threadgroup* memory has performance benefits over `device`.

Additional address spaces are `threadgroup_imageblock`, `ray_data`, and `object_data`, but those are not discussed in this thesis. The MSL language specification makes no clear statements about the exact location of the data or the presence of caches for specific address spaces. This is likely because Metal supports a variety of devices and architectures for which the answer may vary.

2.4.5 Thread grids and Thread groups

Submitted kernel functions execute over `N`-dimensional grids of threads. Metal Shading Language supports grid dimensions of one up to three, where an instance of a kernel function is executed for each point in the grid and referred to as a thread. [38] Threads are organized into threadgroups and executed together, and each thread can be identified by its position in the grid. [34] Metal dispatches threadgroups to different processing elements on the GPU and executes them in parallel. [39] Threads within threadgroups are organized into one-dimensional *single-instruction, multiple-data groups* (SIMD groups), sometimes referred to as *warps*. [34] The composition is shown in the 2.3 figure. The number of threads within a SIMD group is hardware-dependent and can be accessed by reading the `threadExecutionWidth` property. [34] On all Apple GPUs, it is equal to 32. [43] Metal documentation illustrates a two-dimensional grid applied for image processing and a threadgroup formed on the grid, as shown in the figure2.4.

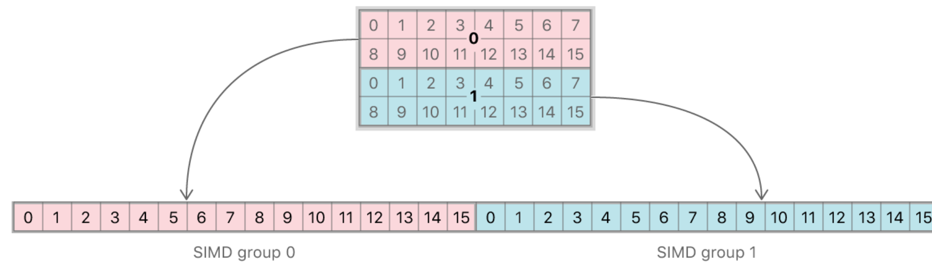

Figure 2.3 Apple Metal threadgroup split into two 16 thread SIMD-groups [34]

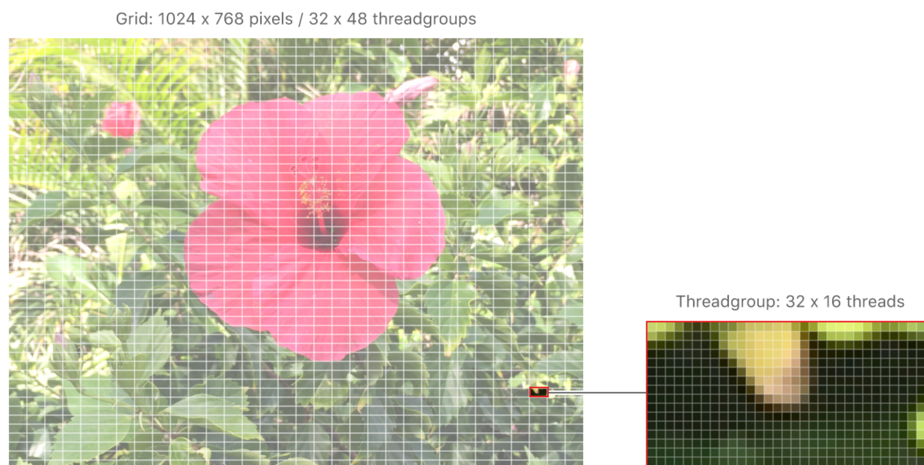

Figure 2.4 Apple Metal thread grid and threadgroups[34]

2.4.6 Metal work submission

This section describes the needed components and the necessary steps to schedule and run Metal compute kernels on the GPU. Metal also allows for render passes, but this section and thesis are restricted to compute passes. Work submission using the Metal API relies on utilizing the API's constructs, which are briefly introduced in the list below [44], [39]:

- *Device* - The GPU device is thinly abstracted by a `MTLDevice` object, which represents the GPU and allows scheduling commands to this device.
- *Library* - `MTLLibrary` represents a collection of Metal shader functions. Contains compiled MSL source code.
- *Function* - `MTLFunction` represents a public shader function in a Metal library.
- *Pipeline objects* - Encapsulates a Metal shading language function. `MTLComputePipelineState` contains a compiled compute pipeline and is a reusable object used through the code.
- *Command Queue* - To schedule work on the GPU device represented by `MTLDevice`, the `MTLCommandQueue` object is created and utilized.
- *Command Buffer* - To issue commands to the GPU device, a Command Buffer is used. The buffer is filled with commands to be executed and later committed. In code, this refers to the `MTLCommandBuffer` and can be created by a specific `MTLCommandQueue` command queue object. The command buffer can then be committed to the GPU and awaited by the host until its completion.

- *Command Encoder* - Each command to be executed is written to the `MTLCommandBuffer` Command Buffer by a Command Encoder. The encoder encodes the commands and includes all data necessary for the GPU to process the task. This is handled by the disposable `MTLCommandEncoder` class. Depending on the type of work to be submitted, a subclass of `MTLCommandEncoder` is responsible for encoding. Those split into:
  - `MTLRenderCommandEncoder` - Responsible for render commands.
  - `MTLComputeCommandEncoder` - Responsible for parallel computation commands.
  - `MTLBlitCommandEncoder` - Responsible for resource management commands.

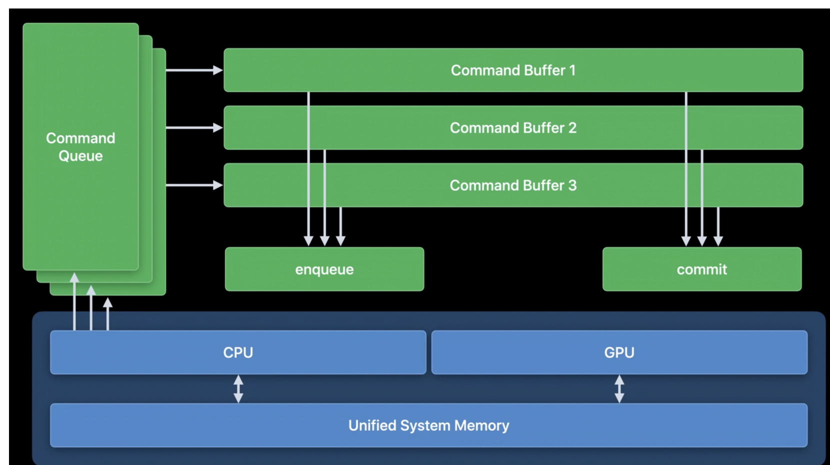

Figure 2.5 Apple Metal Compute submission on M1. The diagram shows the workflow for submitting work to the GPU by committing or enqueuing (to be committed later) the commands. [42]

The typical workflow for submitting compute work to the GPU using Metal is as follows. First, a persistent `MTLCommandQueue` object is created by the `MTLDevice` representing the target GPU device. For each batch of work to be submitted, the command queue creates a disposable command buffer `MLTCommandBuffer`. For each command to be executed within this batch, the command buffer creates a disposable command encoder `MTLComputeCommandEncoder`. This object is responsible for encoding individual input parameters of the kernel and specifying the grid and thread groups. The kernel to execute is determined by passing a persistent pipeline object `ComputePipelineState`, which encapsulates the associated shader code to the encoder, and the arguments are specified and set. To define how the kernel should be executed is specified by the `dispatchThreads` method, which takes as arguments the grid size and thread block size and automatically splits, schedules, and distributes the grid across compute units. This is the case for hardware that supports non-uniform threadgroup sizes, and variable-sized thread groups are automatically created to avoid execution outside of boundaries. Non-uniform threadgroup sizes are supported in the Metal3 GPU family devices. [26] Different methods must be used for hardware that does not support this functionality. When dispatching work using this method, the shader code does not have to check for out-of-bounds access based on the thread position in the grid. Once the arguments are specified, the encoding ends, and the process can be repeated for additional kernels that should be executed within this command buffer. Once all the commands are encoded, the command buffer can be committed by calling the `commit` method on the command buffer. At this point, the GPU will start executing the commands. This process can be repeated by creating an additional command buffer. The execution happens asynchronously to the host code and can be explicitly awaited. [44] Metal promises that the *perceived* order in which commands are executed matches that in which they were ordered, possibly reordering the

commands for performance benefit. [44] Parallel work submission in CUDA requires working with CUDA Streams. In Metal, parallel submission of independent work can be achieved by utilizing multiple command queues. This increases the change of the GPU receiving and processing work as other host threads are not stalled. [43]

2.4.7 Apple GPUs

While Metal is a more generic framework that supports large quantities of hardware, from hardware found in Apple mobile devices, tablets, and non-Apple GPUs found in older Macs [26], the Metal implementation in this thesis focuses on current Macs with newer Apple GPUs, specifically M1 and M2 chips. One of the advantages these chips offer is a unified memory between CPU and GPU. Metal allows the CPU and GPU to read and write the same memory, which potentially brings performance benefits as costly transfers of data between system RAM and video RAM can be avoided. [42] Note that this is considered a costly operation on CUDA. Unfortunately, not much information is publicly available about the architecture and hardware specifications of Apple M1 or M2 GPUs. For a more precise definition, the GPUs targeted in this thesis belong to the Apple8 and Apple7 families, which contain the M1 and M2 chips. In addition, a Metal GPU family is defined, with the target being Metal3, which also includes the M1 and M2 chips but also includes Intel Iris, AMD Vega, AMD 5000-series, and others, as well as other Apple chips intended for mobile devices such as A14, A15. [26] Focusing on Apple7 and Apple8 allows the implementation to rely on the broadest feature set available at the time of writing.

Apple Silicon GPUs contain separate caches for buffer and texture reads. The application may see performance benefits by storing a portion of its resources in buffers and textures, as data read from the cache can be accessed with lower latency than system RAM, reducing stalls or delays. Kernels utilizing more considerable amounts of buffer data may suffer from reduced performance due to the data not fitting into the cache, requiring waiting for system RAM. This limits the reads to system memory bandwidth over on-chip cache bandwidth. Utilizing both texture and buffer cache increases the amount of cache space. [42]

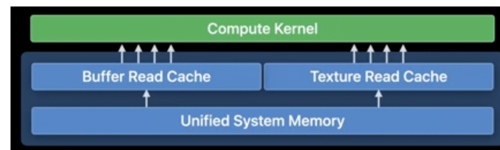

Figure 2.6 Apple M1 GPU cache [42]

Chapter 3
Sequential and OpenMP implementations

This chapter describes the implementation details of the sequential and OpenMP-based implementations. The chapter provides a high-level overview of the general code structure and describes certain design decisions, the reasons leading to these decisions, and their consequences. The usage of the created code is described alongside the discussion about structure. The chapter is finished with performance measurements comparing created variants and collected observations from profiling the code.

3.1 Sequential implementations

The sequential implementation provides Pollard's Rho and ECM based on Weierstrass curves as well as extended Twisted Edwards curves with $a = -1$. It utilizes the GMP library for calculation and holding data. The GMP type utilized is the `mpz_t`, which allows for storing arbitrary precision integers. No other libraries are used in the sequential version.

3.1.1 Pollard's Rho

The first algorithm implemented was Pollard's Rho algorithm. The implementation does not have a complicated structure. The entry point for the algorithm is a function `prho_factorize`, which takes the output argument to which the result is written, the number to factor, an inner function `g` of the algorithm that defines how to exponentiate and add mod `n` to a passed number `x`, and finally, the maximum number of attempts for the algorithm to try before giving up and returning without finding a result.

Code listing 3.1 Pollards Rho factorization function.

```
void prho_factorize(mpz_t rop,
                    mpz_t n,
                    std::function<void(mpz_t, mpz_t, mpz_t, mpz_t)> g,
                    uint64_t max_attempts)
```

This function sets up the starting variables and handles the calling of a low-level factorization function `prho_factorize_direct`, which performs the computation given the decided starting variables. First, the function tries with a simple choice of `x, y` and random `adder`, which is followed by randomly chosen values if the attempt does not succeed. This process of randomly chosen arguments and attempts is repeated until the number of attempts reaches the specified

maximum. The low-level function performs a computationally intensive loop of Pollard's Rho algorithm and returns either `-1` on an unsuccessful run or the factored number if found.

3.1.2 Lenstra's Factorization

The ECM implementation follows a similar pattern to Pollard's Rho. The entry point is once again a function which is called `lenstra_factorize` or `lenstra_factorize_ete` for extended Twisted Edwards (`_ete` being a reappearing suffix for this version of the algorithm. This version will be referenced as *ETE* in the text), which takes the destination `mpz_t`, another `mpz_t` with the number to factorize, the maximum number of attempts (which in this case is tied to the number of different Elliptic curves) as well as an integer lower and upper boundary for point multiplication loop. Both implemented versions of ECM share the same parameters for high-level functions.

Code listing 3.2 ECM factorization function.

```
void lenstra_factorize(mpz_t rop,
                       const mpz_t n,
                       uint64_t max_attempts,
                       int32_t lower_boundary,
                       int32_t upper_boundary)
```

In the Weierstrass version, a new elliptic curve is randomly chosen in each attempt. The process of generating such a curve is first by selecting the variable a in a loop starting from two and incrementing for each attempt. This is followed by randomly choosing a point P. These two variables are now fixed, and the second variable b of the curve is computed from them. If this is successful and a valid elliptic curve can be formed, the computation enters the computationally intensive loop with point kP multiplication. Otherwise, a is incremented, and the search for an elliptic curve continues. For *ETE* version, the flow is more straightforward, as a random point P is generated, and with determined $a = -1$, it is verified whenever a valid elliptic curve can be formed over the point. The lower and upper boundaries determine the value of k, computed as a product of primes smaller than B, which is taken from values between the specified boundary. The purpose of these boundaries is for convenience as they allow the computation to continue even if the initial choice of lower bound did not produce a factor, potentially reducing repeated calls of the function. Initially, the value of B matches the lower boundary. The kP multiplication is computed, and if no result is found, B is doubled until it reaches the upper boundary.

The Weierstrass computation loop iterates over elliptic curve point multiplication, handled by a `lenstra_group_point_mul` function. This function's output $Q = kP$ is then evaluated. If the resulting Q_z is greater than 1, its gcd with the factored number n is returned as the factor. Otherwise, the computation continues with another iteration. The code is similar for *ETE* with a different `lenstra_group_point_mul_ete` function being used to compute the desired results. The gcd of n and both Q_x and Q_z coordinates of the computed point are calculated. If the result is greater than one, a factor has been found.

3.2 OpenMP based implementation

As was the case for sequential implementation, the supported algorithms are Pollard's Rho and Weierstrass and ETE for Lenstra's ECM factorization. The parallel solution on the CPU closely follows the sequential. In addition to the GMP library utilized for the computation and storage of integers, the OpenMP library is utilized to achieve parallel computation in arbitrary precision. Serval variants of each algorithm have been implemented, differing in both the OpenMP mechanisms used to achieve the parallelization and in the strategy chosen to pick starting parameters for the algorithms.

The approaches toward parallelizing the algorithms can vary. For example, in Dvorak's thesis [45], which focuses on CPU parallelization with multiple processes, the inner computational loop over k for Lenstra's algorithm is parallelized and distributed between threads for local copies of P. The approach towards parallelization in this implementation is more simplistic as it will be compared to and serve as a foundation for the GPU implementation. The implementation attempts to benefit from parallelization by running multiple instances of the algorithm with different curves and starting points for ECM or different parameterizations for Pollard's Rho, which increases the probability of finding a solution. [9] This is reflected in both the CPU parallelization and GPU parallelization, where in both cases, instead of parallelizing a portion of the algorithm, many instances of the algorithm are running in parallel in distinct threads, with one finding a fitting solution resulting in the termination of the others.

3.2.1 Lenstra's Factorization

The Weierstrass OpenMP-based implementation has been implemented in five different versions while *ETE* in two. It should be noted that the *ETE* version was introduced much later during the development. In contrast, the initial Weierstrass version was already available for CPU, CUDA, and Metal. Hence, the insight gained from the earlier development was used to limit the number of unnecessary versioning, which would further slow down the implementation. The individual versions combine OpenMP tasks and parallel regions with different starting point selection strategies. Despite the number of versions, the structure of each version is similar to the others and utilizes the same low-level functions for point multiplication and addition on elliptic curves. Those are slightly modified from the sequential implementation, as additional logic is present for verifying if any other thread found a solution to enable quick termination. To produce a result with minimal latency, the competing threads need to terminate quickly once any of them find a solution. This is achieved by reading a shared atomic boolean in the computationally intensive loops. The primary concern with this approach is the balance between the potential overhead by too frequent atomic reads and slower termination on success. This logic is implemented in the two computationally intensive functions of the algorithm. One in multiplication, where the read happens every few iterations, and in the calling loop, where this is verified every iteration. The number of iterations between reads could be further optimized.

Each version is implemented in a distinct factorization function with the version indicated in the name. These high-level functions are meant to abstract away the complexity of the underlying setup required for lower-level functions. The example below shows the function for version two:

Code listing 3.3 OpenMP based factorization function.

```
void omp_lenstra_factorize_v2(mpz_t rop,
                              mpz_t n,
                              uint64_t max_attempts,
                              uint32_t num_threads,
                              int32_t lower_boundary,
                              int32_t upper_boundary)
```

Each version is briefly described in the following list. Note the missing *V1* version. Pollard's Rho algorithm was implemented first, and an attempt was made to make the versions consistent even between algorithms to make the implementation less confusing. In Pollard's Rho, the *V1* version exists but has been observed as less reliable and was not considered for Lenstra. Expecting similar results, this version has been skipped for the ECM implementation, and the versioning started at *V2*.

- `omp_lenstra_factorize_v2` - This version creates OpenMP tasks in batches. Those tasks are distributed between threads until the batch is exhausted, after which a new batch is created, repeating until a result is found. This version can potentially starve threads as they await

the completion of other running tasks until a new batch is created. The variable a is chosen randomly in each task.

- `omp_lenstra_factorize_v3` - The third version is also OpenMP task-based. In this version, the tasks are not created in batches, and their creation is left to OpenMP without any interference. This resolves the issues in the second version in which threads could be underutilized but potentially come with an overhead with many unnecessarily created tasks. As in the second version, the variable a is chosen randomly in each task.

- `omp_lenstra_factorize_v4` - The fourth version simplifies the parallelization by dropping tasks and relying simply on parallel regions. This approach resolves issues observed in previous versions, as in this version, each thread that fails to find a solution given a specific elliptic curve restarts the process with a new curve. Threads then loop and create new curves until a solution is found. The variable a is chosen randomly in each task.

- `omp_lenstra_factorize_v5` - The fifth version is based on parallel regions and differs in the selection of variable a, which is now sequentially incremented for each attempt in a thread. The same applies to the P_x coordinate, which is sequentially incremented but comes from a distinct interval per thread. This seemingly arbitrary choice of parameters reduces the likelihood of generating the same starting parameters, which, however unlikely, could be a problem in the previous version.

- `omp_lenstra_factorize_v6` - In the last version, again parallel regions are utilized. The value a comes from a distinct interval per thread, with point P being generated randomly. Having a distinct value of a per thread should not generate identical elliptic curves.

- `omp_lenstra_factorize_v4_ete` - The $V4$ of the ETE version follows the structure of the Weierstrass $V4$ version, but relies on the ETE low-level functions. With P being generated randomly.

- `omp_lenstra_factorize_v5_ete` - Same as with the previous version $V5$ of the ETE version follows the structure and strategy of the Weierstrass $V5$ version. The x coordinate of p is selected from an interval, and y is randomly generated.

In summary, the first versions focused on selecting the appropriate OpenMP parallelization construct. In contrast, further versions focused on initial parameter selection and either reducing or eliminating potentially duplicate elliptic curves that would result in wasted computations. Given the size of the numbers to factor and the small probability of generating the same parameters, it's questionable whether adjusting the parameters to avoid duplicate elliptic curves is beneficial. This should become apparent in the measurements later.

3.2.2 Pollard's Rho

In total, seven implementations of Pollard's Rho have been created. These, as was mentioned, combine different strategies for selecting starting arguments as well as OpenMP constructs for parallelization. Comparing the versions with the version in the Lenstra implementation, it can be seen that the versions are consistent in terms of OpenMP constructs and, naturally, differ in parameter selection. The versions are described in less detail, as the reasoning behind each version was specified in the previous section. In summary, initial versions focused on finding a good pattern for concurrency, while later versions experimented with parameter selection and potentially reducing or eliminating redundant computations.

- `omp_prho_factorize_v1` - Uses OpenMP parallel for construct with the variable a being selected from a distinct interval and x being selected randomly.

- `omp_prho_factorize_v2` - Utilizes OpenMP tasks created in batches. As in *V2* of ECM, a batch of tasks is created, and until all tasks are completed, no new batch is created. The first fifty attempts per thread are made with small $x = y = 2$ and random a. After that, the implementation switches to all variables being chosen randomly.
- `omp_prho_factorize_v3` - Tasks are no longer restricted to batches in this version. The variable selection strategy is the same as in the previous version.
- `omp_prho_factorize_v4` - This version switches to parallel regions. The variables x and y are set to two, and a is chosen randomly per each attempt in each thread.
- `omp_prho_factorize_v5` - Based on parallel regions, with x and y set to two and a being chosen from a distinct interval for each thread and incremented for every attempt.
- `omp_prho_factorize_v6` - Utilizes parallel regions, a is set to one and x and y are incremented per-attempt from distinct intervals per-thread.
- `omp_prho_factorize_v7` - Utilizes parallel regions with all parameters a, x, y being chosen incrementally from distinct regions per-thread.

3.3 CPU Versions comparison

In order to reduce the number of versions for further comparison with GPU-based implementations and decrease the complexity and time requirements for future measurements, an initial set of measurements has been made on the implemented parallel CPU versions. The measurements have been made on an ARM64 device. The ARM device has an 8-core Apple M2 chip. These measurements are not as detailed as the final comparison and have been made on a smaller subset of parameters and inputs. Still, given the apparent advantage of some of the implementations, this comparison allows for the discarding of the less efficient implementations from further evaluation and allows for greater attention to the more promising variants as the comparison extends to GPU implementations. This measurement, for example, does not observe how each implementation scales with a different number of threads. Its purpose is to reject the sub-optimal implementations. The comparison has been made on the numbers listed below.

- *410,008,714,444,926,584,643,751,636,103* - 99 bits
- *740,823,820,721,940,713,928,228,049,555,961* - 110 bits
- *107,086,883,892,938,461,277,930,808,325,667,887,273* - 127 bits

Those numbers provide sufficient difficulty for the algorithms, allowing for meaningful comparison while not taking an overwhelming amount of time to factor. They contain 30, 33, and 39 digits, respectively.

3.3.1 Comparison

The measurements were made with four and eight threads, and to reduce the effect of randomness in the parameter selection, each version was rerun thirty times for the first two numbers. In contrast, the measurements of the last number took significantly more time to complete and were measured ten times. The compiled binary has been produced with the following optimization flags using the GCC 12.2.0 compiler for ARM64:

Code listing 3.4 Optimization flags used for OpenMP variants.

```
-O3 --host=aarch64-apple-darwin
```

The results can be visually compared in the 3.1 figure. The versions *V2* and *V3* of Lenstra's algorithm provided very good results for the first and smallest integer but struggled to find a factor for the second and third integers. Instead, they frequently exhausted the maximum number of attempts and then terminated. The versions *V4*, *V5*, and *V6* produced very similar results across all three measured numbers.

The inconsistent behavior for the *V2* and *V3* versions of Lenstra discourages their further usage. In contrast, *V4*, *V5*, and *V6* consistently provided relatively good results across all measurements, making them good candidates for further evaluation. Overall, the *ETE* versions dominated over Weierstrass versions. In the case of Pollard's Rho, the *V5*, *V6*, and *V7* versions provided consistently good results but were frequently beaten by some of the more random counterparts. In Pollard's Rho, The other versions produced relatively close results, making the comparison more difficult. Overall, the *V4* version seemed to perform quite well for the largest number but was otherwise unremarkable. It must be stated that better performance for large numbers is favored over smaller ones.

3.4 Measurement summary

The *V4* provided good results across platforms and will be further utilized. As per versions *V5* and *V6*, those produced very similar results. From the measured data, there is no clear winner for Weierstrass versions. The *V4* version will be favored as it is simpler. Versions attempting to prevent potential conflict in chosen parameters did not show improvement over those that did not. Since the *ETE* versions provided the best but similar results, only the *V4* will be considered further. For Pollard's Rho, the *V4* version representing random selection with parallel regions will be picked. For a clear overview, the following versions will be evaluated further:

- `omp_prho_factorize_v4`
- `omp_lenstra_factorize_v4_ete`

It should be noted that some of the versions relying on distinct intervals do not necessarily benefit from the increased thread counts, as the number of threads directly affects the parameter choice, leading to potentially more favorable starting parameters. This behavior can be observed in the 3.2 figure, which shows the time until a solution has been found by the *V7* version of Pollard's Rho. Still, more threads generally improved the results rather than not. Similar observations can be made for the purely random variants. As their nature is inherently random, the results can significantly vary depending on how *lucky* the random choice of parameters is. This section's measurements have been restricted to keep measurement time sensible for the CPU versions. Some of the unimpressive versions have been removed from further evaluation, which should also have a larger selection of parameters.

3.5 Profiling the CPU implementation

This section will provide some insight into the performance bottlenecks of the CPU implementations. The profiling will be conducted on the ARM platform, focusing on the implementation bottlenecks rather than algorithmic or parameter-specific performance impacts. The profiling will be conducted using the Instruments application for MacOS.

When inspecting Pollard's Rho algorithm in its sequential implementation, it can be observed that the heaviest impact has the GMP modular reduction and GCD functions. This can be seen in the 3.3 figure showing the output of the Instruments utility. This behavior is consistent between runs and, given the relative simplicity of Pollard's Rho implementation and complexity of the mentioned operations, not greatly surprising. This repeats in the parallel implementations

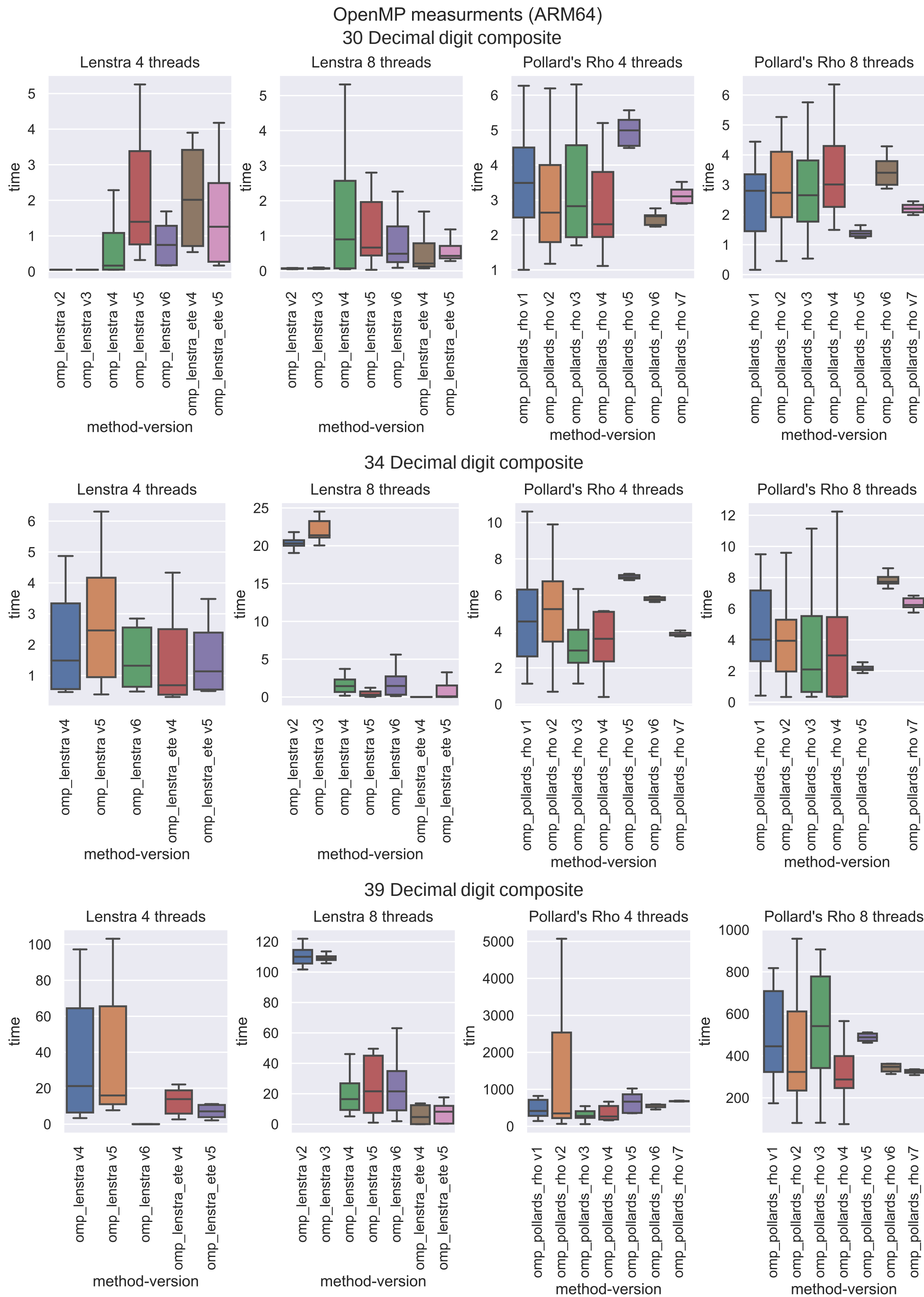

Figure 3.1 Measured runtimes per algorithm and version with eight threads (ARM64).

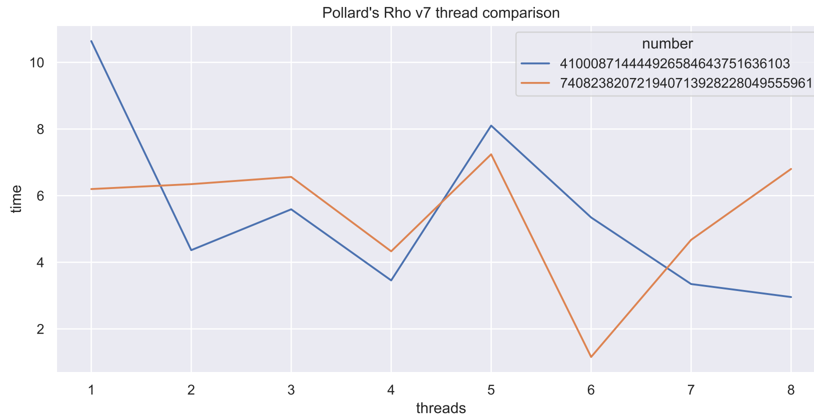

Figure 3.2 Time until solution for v7 version of Pollard's Rho for different thread counts (ARM64 M2).

as well. The majority of the time is spent computing modular reduction and GCD consistently on all threads.

For Lenstra's implementation, which can be seen in figure 3.4, most of the weight is spent on the GMP extended gcd function. However, with the computation being more complex, the rest of the time is more evenly distributed across the remaining operations. The second most dominant is modular reduction, which is closely followed by subtraction. This is not surprising, as touched on in the *Algorithms* chapter. Computing the modular inversion is a computationally very expansive operation, and using alternative approaches, such as projective coordinate representation, that do not require finding inversion is generally favored. Suppose similar attention is given to the *ETE* version. In that case, it is clear that the time is now more distributed across different operations, with the modular reduction being the most costly. Overall, most time of the computation is spent in the GMP functions, which are already highly optimized and over which the implementation has little control.

Weight		Self Weight	Symbol Name
41.28 Gc	100.0%	280.38 Mc	prho_factorize_direct(__mpz_struct*, __n
19.10 Gc	46.2%	1.67 Gc	__gmpz_gcd factorization-main
13.65 Gc	33.0%	757.45 Mc	__gmpz_mod factorization-main
6.17 Gc	14.9%	103.62 Mc	prho_g(__mpz_struct*, __mpz_struct*, .
1.69 Gc	4.0%	1.51 Gc	__gmpz_sub factorization-main
225.11 Mc	0.5%	225.11 Mc	__gmpz_cmp_ui factorization-main
163.68 Mc	0.3%	163.68 Mc	std::_Function_handler<void (__mpz_st

Figure 3.3 Heaviest stack trace for sequential Pollard's Rho implementation.

3.6 Summary

This chapter, being the first to discuss the actual implementation created within this thesis, focused on the CPU-based implementations in their sequential and OpenMP variants, showcasing and evaluating various implemented versions. Observations were made into the performance characteristics of the implementations, which seemed to align with the theoretical understanding. The most favored variants, $V4$ version for both Pollard's Rho and ECM, were selected and will

Weight		Self Weight	Symbol Name
7.62 Gc	100.0%	2.01 Mc	⌄lenstra_factorize(__mpz_struct*, __mpz_struct
7.62 Gc	99.9%	59.66 Mc	⌄lenstra_group_point_mul(__mpz_struct*, __m
7.53 Gc	98.8%	431.46 Mc	⌄lenstra_group_law_add_point(__mpz_struc
3.85 Gc	50.4%	38.21 Mc	> __gmpz_gcdext factorization-main
1.02 Gc	13.3%	84.73 Mc	> __gmpz_mod factorization-main
916.00 Mc	12.0%	265.80 Mc	> __gmpz_sub factorization-main
736.98 Mc	9.6%	408.40 Mc	> __gmpz_mul factorization-main
314.22 Mc	4.1%	38.24 Mc	> __gmpz_mul_si factorization-main
105.20 Mc	1.3%	44.05 Mc	> __gmpz_add factorization-main
53.71 Mc	0.7%	24.41 Mc	> __gmpz_set factorization-main
32.83 Mc	0.4%	32.83 Mc	__gmpz_cmp_ui factorization-main
24.13 Mc	0.3%	24.13 Mc	__gmpz_cmp factorization-main
23.41 Mc	0.3%	23.41 Mc	__gmpz_clear factorization-main
13.85 Mc	0.1%	13.85 Mc	__gmpz_init factorization-main
13.26 Mc	0.1%	13.26 Mc	DYLD-STUB$$free factorization-main
1.00 Mc	0.0%	1.00 Mc	__gmpz_pow_ui factorization-main
1.00 Mc	0.0%	1.00 Mc	__gmp_default_free factorization-main
14.51 Mc	0.1%	4.00 Mc	> __gmpz_set factorization-main
11.75 Mc	0.1%	2.00 Mc	> __gmpz_init_set_si factorization-main
1.00 Mc	0.0%	1.00 Mc	__gmpz_init factorization-main
1.00 Mc	0.0%	1.00 Mc	__gmpz_clear factorization-main
1.00 Mc	0.0%	1.00 Mc	__gmpz_cmp_ui factorization-main

Figure 3.4 Heaviest stack trace for sequential Lenstra implementation.

provide a decent reference for further work on the CUDA and Apple Metal implementations and a more detailed comparison in the future.

Chapter 4

Multi-precision integer arithmetic on Apple Metal

As discussed in the previous chapters, various libraries allow arbitrary precision arithmetic on the GPU. Those, however, only target CUDA and seem to mostly attempt to parallelize the arithmetic operations. While the factorization algorithms could benefit from potentially faster arithmetic, this thesis explores a different approach, where many competing instances of the algorithm seek solutions from different starting points independently. That is, there is no thread cooperation to compute the results. No widely available library existed that would enable this form of computation in Metal at the time of writing this thesis. Hence, a new library allowing for multi-precision integer arithmetic was implemented for usage with Metal and CUDA. The name for this library is Metal Arbitrary Precision Library, or, shortly, MAP Library. Initially, it was intended only for Metal but was later ported to CUDA as well. This implemented library covers a subset of functions and types compared to the GMP library and can't compete with the functionality and optimizations offered by GMP but allows the parallelization of the selected algorithms. This chapter is dedicated to describing the library in detail.

4.1 Metal Arbitrary Precision library

The library's implementation scope in this thesis was narrowed to enable the required functionality for parallelizing the algorithms on the GPU. It was created specifically to work with the restrictions of this environment. The creation of such a library requires substantial effort, but its usefulness ranges beyond the narrow topic of integer factorization discussed in this thesis and can be utilized for any multi-precision integer arithmetic usage that may arise. The supported integer arithmetic in this library is single-threaded, where each GPU thread is intended to run its own computations, which means that the individual arithmetic operations are sequential and intended to be run independently in parallel. Shortly, the threads do not cooperate to produce results, giving some contrast to the previously mentioned CGBN library. In the context of factorization algorithms, this means that multiple parallel instances of the algorithm are running, each requiring its own arithmetic operations. This is done as an alternative over a single or reduced set of instances with parallelized operations.

4.2 Storing arbitrary precision integers

The representation of numbers closely reassembles the GMP and LibTomMath implementations, with integers being represented as a struct of three elements (in the case of LibTomMath, the sign is stored separately as an additional element). The first two signed integers indicate the total length and number of used limbs. The third is a pointer to an array of unsigned integers, where each element stores a portion of the number in absolute value, with the least significant portions of the numbers being stored first at lesser indexes. The integer storing the number of used limbs is signed, with the sign determining the overall sign of the value stored in this struct. The definition of this structure in the Metal Shading Language can be seen below.

Code listing 4.1 MAP Library dynamic sized integer structure.

```
typedef struct map_int{
    map_digit len = 0;
    map_digit used = 0;
    device map_word * numbers = nullptr;
} map_int;
```

The name `map_int` has been chosen for the structure; it stands for Metal Arbitrary Precision Integer. The size of unsigned integers can be selected during compilation by varying definitions of `map_word` with the verified options being 32-bit and 16-bit unsigned integers. The same compile-time choice is present for the sign integers represented by `map_digit` offering 32 and 16-bit choices. In general, the choice of limb size is significant, as performance can be improved by selecting sizes favored by the hardware. Choosing a smaller size might be beneficial on particular hardware.

A number of the implemented arithmetic operations require more precision during a portion of the computation. For example, during the multiplication of two 32-bit limbs, a carry might be produced and needs to be passed further in the algorithm. Using a 64-bit destination for this computation makes this straightforward. Hence, additional data types need to be specified to satisfy this requirement. For 32-bit limbs, an additional type `map_word_larger` is defined as a 64-bit unsigned integer. Meanwhile, for 16-bit limbs, a 32-bit unsigned integer is used. On a related note, the LibTomMath library prefers to use 64-bit limbs, or digits `mp_digit` as referred to in the library. Digits of this size are potentially impractical on Metal, as for some operations, larger-than-used digit variables are needed, while the Metal Shading Language 3.0 language reference shows only scalar data types of size up to 64 bits. Choosing 64-bit limbs would make the implementation of certain sections of the library much more complicated. Hence, there is a choice of 32-bit and 16-bit options.

The integer value stored this way can grow by using additional allocated limbs. This is reflected by the `used` integer incrementing until it matches the `len`, indicating no more allocated space to store the value. This requires copying the value to another instance with sufficient space, which has to be done on the CPU. The struct is declared separately for the CPU and GPU, as the Metal code requires address space specification for pointers and references. [38] While the `map_int` instances live in thread address space, the memory where the digits are stored is in the device address space and is allocated and passed to the shader from the CPU. The address space for the defined struct cannot change in Metal.

4.3 Memory allocation limitations in Apple Metal and impact on implementation

The Metal Shading language does not support memory allocations in GPU code. [38] Metal shading language also does not allow for variable length arrays. [38] With the arithmetic being intended to be performed by the GPU, this would mean that if insufficient space has been

allocated beforehand, the operation may lack space to store the valid result. One limitation brought by the lack of dynamic allocation can especially be felt in more complex functions of the library, which require additional temporary `map_int` variables to produce a result beyond the usual in-out variables. This forces either a requirement that all variables appearing in the function be dynamically allocated beforehand and passed to each function and overall to the shader, or an alternative approach would be to have compile-time size-defined helper variables within each function requiring temporary variables. As was previously mentioned, a similar approach could be seen in [9], where the size was determined at compile time for all numbers. This is a somewhat rigid approach, but it has its benefits, as it allows for additional optimizations such as loop unrolling.

Unfortunately, this is further complicated by strict rules for address spaces in Metal. In MSL, each pointer requires address space specification, which can not be changed, with local variables being by default in the thread address space. [38] If thread address space statically-sized arrays were considered for limbs in shaders, it would cause conflict with the device address space as declared in the `map_int` struct, which, during instantiation, has its data as passed to the shader. Metal shading language explicitly prohibits dynamic or static casting to different types, such as shown in the example below.

Code listing 4.2 Prohibited casting in MSL.

```
uint32_t tmp_helper[FIXED_HELPER_LEN_GPU] = { 0 };
static_cast<device int32_t *>(tmp_helper);
```

This forces the use of dynamically allocated and passed helper variables, as introducing compile-time size helpers in combination with dynamically sized inputs, although possible, is impractical and complicates the code. An alternative would be to use a constant size for all multi-precision integers. This approach was not initially chosen as it was deemed too restrictive for more generic library use but was later revisited for performance reasons. Unfortunately, helper variables being passed to the functions that need them complicates code, increases the number of parameters in functions, and makes it more challenging to use. However, it is necessary to address the limitations of the Metal Shading Language while allowing varying input sizes. These limitations result in design choices that differentiate the implemented arbitrary precision library from its existing CPU-based counterparts, such as GMP or LibTomMath.

The library allows working with dynamically sized multi-precision integers, but this size can not be changed within a GPU shader. In practice, this results in a reduced number of options for GPU code and computations. One option is that sufficient memory is allocated before the GPU shader runs, which may result in significant memory overhead and may be hard to estimate depending on the use case. The second option is that for multiple concurrently running computations, it is accepted that some may reach incorrect results or that their computations are corrupted due to insufficient memory, and this state is accepted. Another possibility is the combination of these approaches, where failure is permitted but should be recovered from. Corrupted results are detected after each GPU phase of the computation, more memory is allocated, and the original values are restored to the newly allocated memory. This then allows the code to repeat the last phase of computation, ensuring the correctness of the results. This approach will result in all parallel computations being correct. The memory footprint (the size of buffers being directly passed to the shaders) may be less for a portion of the runtime than if the memory was allocated beforehand, but this is likely negligible compared to the performance cost of copying and the memory footprint of keeping a separate backup of the computational data. However, this approach allows arbitrary-precision computation to provide expected and valid results if required. This costly approach may be the only option in computations where the expected result size is unknown, but corrupted results are not tolerated.

The implemented algorithms in this thesis utilize the second approach, where some failure is tolerated but should be minimized by allocating sufficient resources. The nature of the problem allows it so that the largest possible number size can be estimated from the factored number.

The factorization implementations allow allocating varying amounts of memory to avoid failure due to insufficient memory. The implemented algorithms reset the validity of the computation between attempts.

4.4 Fixed-size integer representation

As mentioned in the previous section, the initial approach was to implement the library to be fully dynamic-sized despite the limitations of the MSL language. This approach has some negatives, which will be discussed in greater detail in later chapters. Mainly, it prevents some useful optimizations in the library functions and has a high memory bottleneck. In hopes of providing a better, more optimized library, an additional fixed-size representation was introduced alongside the dynamically sized section. The representation is referred to as `map_int_f`, suffixed with `_f` for *fixed*. The maximum size is defined during compilation and cannot be changed later or combined with the dynamically sized implementation. The library matches the functionality offered for this representation in the GPU section of the library but has no support for the CPU functions, where there was little need for it (conversion between fixed and dynamically sized `map_int`s is a straightforward operation on the host).

The `map_int_f` definition can be seen below. One of the benefits is that some memory is saved as the library no longer needs to store the information about the allocated size. This is a greatly appreciated benefit of this representation in the highly memory-constrained environment each GPU thread has, potentially reducing register pressure. An additional difference that can be observed is the fixed size permits changing address space from *device* to *thread*, which requires fixed-size arrays.

Code listing 4.3 Redefined MAP library structure for fixed-size integers.

```
typedef struct map_int_f{
    map_digit used = 0;
    thread map_word numbers [MAP_FIXED_SIZE];
} map_int_f;
```

4.5 Metal Arbitrary Precision library function conventions

The library attempts to provide a clear and consistent interface across the functions it provides. For this reason, a set of conventions is defined to ease the usage of the library and allow predictable behavior. All functions implemented for the arbitrary precision function directly manipulating `map_int`s need to be compliant with the following design conventions:

- All `map_int` and `map_int_f` parameters to functions are passed by references.
- The functions, unless specific only to the CPU (such as host functions used to initialize `map_int` variables to a specific value), will not do any dynamic memory allocations or deallocations.
- Each helper variable passed to a function must be unique and different from other arguments. Otherwise, the result is undefined. Helper variables are passed from external storage for temporary computations in complex operations (such as division, gcd).
- If the library function can fail, such as due to insufficient allocated memory in the parameters, the function must communicate any failure with a return boolean value set to false. In such cases, the results are undefined. Function, where failure is not expected to be possible, can have a void return type.

- The order of library function parameters is as follows:
  - The First parameters of the functions are the in-out parameters to which results will be written. These parameters may be used as helper variables during the computation for optimization, and they need sufficient space allocated beforehand for the computation to succeed.
  - The Second group of parameters are purely constant inputs. These form the basis for the operation. For example, those would be the two variables to add in the addition operation.
  - The last group of parameters are helper variables. Those are temporary computational variables with no meaningful input or output value but need to be passed to the function due to memory allocation limitations. Helpers will be written to and need sufficient space allocated for the computation to succeed. The content of these helper variables is not important on the call of the function and is undefined after return.
- By default, the functions should allow specifying calls where the return parameter is also one or more of the input parameters.

$$func(a, a, b)$$

  This pattern translates to $a = func(a, b)$. If this pattern is not supported, it is clearly communicated in the function name with a `_unsafe` suffix. A safe version of the function frequently exists in such cases and is marked with a `_safe` suffix. The safe versions require additional helper variables to be passed in order to produce a result and have a larger memory footprint. The unsafe versions of the function have a smaller memory footprint but are less generic.
- All input parameters that are not output parameters or helper variables are constant, and the functions will not modify them. That is, unless the reference matches any of the output parameters.

  An example of a typical function following the conventions above can be seen below.

$$func(return_value,\ const\ input_1,\ const\ input_2, ...,\ helper_1, ...)$$

The metal version of the functions needs to specify the address space in the function parameters. This is defined as `thread`. That means the library functions are not meant to be the entry point for the GPU code but rather be called by a computational kernel with `map_int` variables stored in the thread address space decoded from its inputs. An example of an MSL GPU function can be seen below.

Code listing 4.4 MAP Library GPU multiplication signature.

```
bool map_mul(thread map_int & rop,
             thread const map_int & a,
             thread const map_int & b,
             thread map_int & helper_a);
```

4.6 Metal Arbitrary Precision library structure

The library contains the definition of the `map_int` and `map_int_f` structures, a holder class described in the following sections, and a set of CPU and GPU functions. The GPU and CPU functions overlap but are not the same. The CPU section has functions related to loading, transforming, and printing the data in `map_int` format, as well as a large section of supported

arithmetic and algorithmic operations. The GPU section also contains a set of functions helpful in initializing and copying a more significant number of `map_int` variables in parallel. This includes random number generation and copying between a more significant number of variables. The library layout is as follows (ignoring specific headers, C++, and metal files). Items related to the CUDA implementation are listed but will be discussed in a later section:

```
map_lib
 ├── map_int
 ├── map_holder
 ├── cap_holder
 ├── CPU
 │    └── CPU functions
 ├── Metal GPU
 │    └── Metal GPU functions
 └── CUDA GPU
      └── CUDA GPU functions
```

4.7 Supported functions

As mentioned above, the library splits the implementation of CPU and GPU functions, which overlap, but the sets of each are not identical. The following list mentions each function with a brief description. To avoid duplicate descriptions, each item indicates whenever the implementation is available for both GPU and CPU or only one of them. As for support for fixed-size integers, all functions available for the GPU are available in variants for both fixed and dynamic sizes, distinguished by `_f` suffix in the function name. It should be noted that the Metal GPU functions defined for fixed-size integers can have different parameters as they do not require dynamically sized helper variables to be passed from the outside (although this was preserved for fixed-size on CUDA). The CPU offers no support for fixed-size integers.

- `map_abs` - Absolute value for `map_int`. *(CPU, GPU)*
- `map_add_positive` - Low-level addition. Adds two `map_ints` as positive numbers. When supplied with a negative, it ignores the negative sign. *(CPU, GPU)*
- `map_add` - High-level addition of two `map_ints`. It supports signed inputs built on top of positive addition and subtraction. *(CPU, GPU)*
- `map_clear` Clears the `map_int` and deallocates memory. *(CPU)*
- `map_clip_leading_zeros` - Sets the value of used limbs based on the first non-zero limb. *(CPU, GPU)*
- `map_copy_3d` - Copies 3 dimensional X, Y, Z `map_int` values in parallel to another set of X, Y, Z values. *(GPU)*
- `map_copy` - Copies one `map_int` to another provided the destination has sufficient space allocated. *(CPU, GPU)*
- `map_div_2` - Faster division of `map_int` by two. Implemented as shift right by `n` digits. *(CPU, GPU)*
- `map_div_unsafe` - Divides one `map_int` by another, produces quotient and remainder. Input and output parameters cannot be mixed. Uses multi-precision division algorithm as defined in [13].*(CPU, GPU)*
- `map_div_safe` - Safe division that supports combining input and output. Uses multi-precision division algorithm as defined in [13]. *(CPU, GPU)*

- `map_eq_abs` - Verifies if two `map_ints` match ignoring the sign. *(CPU, GPU)*
- `map_eq_zero` - Verifies if the specified `map_int` equals zero. *(CPU, GPU)*
- `map_eq` - Verifies if two `map_ints`, or `map_int` and `map_word` match. *(CPU, GPU)*
- `map_flip_sign` - Flips the sign of the `map_int`. *(CPU, GPU)*
- `map_gcd` - Produces the gcd of two `map_ints`. Uses the Binary gcd algorithm as defined in [13]. *(CPU, GPU)*
- `map_gcd_ext` - Produces the gcd of two `map_ints` as well as coefficients of Bézout's identity. Uses the Binary extended gcd algorithm as defined in [13]. *(CPU, GPU)*
- `map_ge` - Verifies if `map_int a` is greater or equal to `map_int b`. *(CPU, GPU)*
- `map_get_string` Given a `map_int` returns a string with the binary representation of the passed number. *(CPU)*
- `map_gt_abs` - Verifies if `map_int a` is greater than `map_int b` ignoring sign. *(CPU, GPU)*
- `map_gt` - Verifies if `map_int a` is greater than `map_int b`, an additional version exists for comparison with a `map_word_larger_signed`. *(CPU, GPU)*
- `map_init_from_map` - Initializes a `map_int` by allocating sufficient memory and copying the content of the passed `map_int`. *(CPU)*
- `map_init_from_string_binary` - Initializes a `map_int` by parsing the passed string with the binary representation of the desired number. *(CPU)*
- `map_init_from_value` - Initializes a `map_int` by setting it to the passed integer value. *(CPU)*
- `map_lt` - Determines if the `map_int a` is lesser than `map_int b`. *(CPU)*
- `map_mod_unsafe` - Produces the modulo residue of two `map_ints`. Input and output parameters cannot be mixed. Slow implementation that relies on division to produce results. *(CPU, GPU)*
- `map_mod_safe` - Produces the modulo residue of two `map_ints`. Allows for mixed input and output parameters. Slow implementation that relies on division to produce results. *(CPU, GPU)*
- `map_mul_2` - Fast `map_int` multiplication by two, achieved by shift left by `n` digits. *(CPU, GPU)*
- `map_mul` - Multiplies two `map_ints` together. Uses the "schoolbook" multiplication algorithm. *(CPU, GPU)*
- `map_reset_to_zero` - Sets a `map_int` value to zero. *(CPU, GPU)*
- `map_set_from_index_md` - Multidimensional copy. Assumes the passed buffer stores multiple `map_ints` with a specified length in between. Each thread copies and sets to its index value from the first `map_ints` in each dimension. *(GPU)*
- `map_set_random_md` - Sets a portion of `map_ints` to random values with variable length. *(GPU)*
- `map_set_random` - Sets a range of `map_ints` to random values. *(GPU)*
- `map_set_thread` - Sets a range of `map_ints` to value of thread indexes. *(GPU)*

- `map_sub_positive` - Low-level subtraction. Subtracts two positive `map_int`s. When supplied with a negative `map_int`, ignores the negative sign. *(CPU, GPU)*
- `map_sub` - High-level subtraction of two `map_int`s. Supports signed inputs. *(CPU, GPU)*
- `map_swap` - Swaps the values of two `map_int`. *(CPU, GPU)*

One of the major disadvantages of the library is that it relies on simpler algorithms for complex, computationally intensive operations, such as multiplication or modular reduction. GMP utilizes up to 7 algorithms to provide optimal performance during multiplication, but the MAP library is limited to a single algorithm. While GMP may elect to utilize various algorithms for different input sizes, the MAP library is limited to one. This is a common pattern across the supported functions and one direction in which the library could be improved.

4.8 Implementation differences between CPU and GPU function versions

During development, the intention was to keep the CPU and GPU functions as similar as possible, mainly to simplify maintenance and development. The library is primarily focused on running on Apple GPUs, and the function design reflects that and is observed in the CPU versions. For example, the CPU version could allocate all needed non-input variables, but to diverge less from the GPU version, it requires these helper variables to be passed from the outside. Developing tests and debugging for the CPU versions is significantly more straightforward and less time-consuming than doing the same for the GPU functions. Developing the CPU versions first made catching bugs early during development significantly easier, after which the functions could be ported to the GPU with less issues. Most of the differences between CPU and GPU versions are in address space specifications on the GPU and relying on the metal standard library over the C++ standard library, such as for computing absolute values of integers.

4.9 Working with a large number of arbitrary precision integers on the GPU

The implementations in this thesis focus on running many parallel instances of the algorithms with different starting points, each instance performing its independent computations. In the OpenMP implementations, the parallel regions trivially manage and allocate their variables with complete independence on other running instances. The overall code structure for Metal implementation is significantly different. Each running instance is in sync with others, and the used variables must be modified synchronously in bulk. Dealing with allocations per variable used across each algorithm and keeping track of the validity of results per instance would result in code that is both difficult to read and maintain. In addition, the Metal API requires that data passed to the GPU is stored in a particular format, the options being textures, MTL buffers, or more sophisticated structures for which Metal handles the passing automatically but enforces passing parameters as a constant. This demands code that allows for easy access and management of variables in bulk for each concurrent instance. A specific class for storing and owning these variables has been created to fulfill this need. The outline of this class can be seen below.

Code listing 4.5 MAP Library integer holder class interface.

```
class MetalArbitraryPrecisionIntegerHolder
{
    MetalArbitraryPrecisionHolder(MTL::Device * device,
                                  int32_t n,
                                  map_digit len,
                                  bool helper = false);
    ~MetalArbitraryPrecisionHolder();

    int32_t members; // the number of member variables stored
    MTL::Texture * properties;
    MTL::Buffer * numbers;
    MTL::Buffer * current_length_buffer;

    map_int Get(int32_t index);
    bool Valid(int32_t index);
    void SetSigned(int32_t index, map_word_signed value);
    void Set(int32_t index, map_word value);
    bool Set(int32_t index, const map_int & value);
}
```

The specified class is referred to as a ‘holder‘, which should allow working with arbitrary precision integers in bulk efficiently. This class is intended to store variables of the same use, such as multiple instances of a specific computational variable kept for each instance of the algorithm, but storing strided variables is also possible. For example, in ECM implementation, one handler class stores multiple coordinates of points on the EC.

The handler class does not store individual `map_int` instances. Instead, it stores the metadata such as validity and used limbs in 1D 32-bit or 16-bit textures and enforces the same allocated length per each stored variable. Since Apple M1 and M2 GPUs have a separate cache for textures and buffers. Storing the data partially in textures and buffers potentially allows for improved performance by better utilizing available cache memory. In addition, storing a set of values as arrays of `map_int` structures may not be the most efficient. The data, or limbs of the variables, are kept in a one-dimensional Metal buffer and accessed individually by index multiplied by the uniform allocated length. Working directly with metal buffers and textures is cumbersome and difficult. The class supports setting individual values by passing integer or `map_int` arguments. It can export individual `map_int` variables by constructing them from the modified storage format, as well as report on the validity of a particular member. The class publicly exposes the stored textures and metal buffer to allow easy encoding for submission to the GPU. Additionally, the class can also be utilized to store fixed-size integers.

4.10 Encoding and using arbitrary precision integers in GPU shaders

A specific process must be followed to get a `map_int` from a handler class into a shader. The shaders utilizing the library have a common pattern for passing the necessary data inside. For Metal, first, a buffer with the limb data is passed, followed by a single buffer with a single value indicating the number of allocated limbs per member and a texture containing properties such as the used number of limbs and validity. The texture is indicated as read-only if the members are not modified.

Code listing 4.6 Example Metal computational kernel MAP library argument passing.

```
    kernel void example_function(
    device map_word * a_numbers [[buffer(0)]],
    device const map_digit & a_current_length [[buffer(1)]],
    texture1d<map_digit, access::read_write> a_properties [[texture(1)]],
    uint16_t index [[thread_position_in_grid]])
{
    auto a_meta = a_properties.read(index);
    map_int a {a_current_length, a_meta[0], &a_numbers[index * a_current_length]};

    bool valid = func(a, ...);

    a_meta[0] = a.used;
    a_meta[1] = valid;
    a_properties.write(a_meta, index);
}
```

For helper variables, parameter passing is simple as no validity tag or persistence of values is needed, and as a consequence, the metadata-containing texture is not passed. This greatly reduces the memory footprint of shaders and provides performance benefits. The individual variables must be created with all the necessary parameters gathered. This is achieved by reading the texture value for the specific thread index and creating a thread `map_int` instance, using the fixed member length, specific property as read from the texture, and address to the buffer section containing the associated member limbs. This is achieved by jumping in the buffer to the desired index (frequently matching the thread index in the implementation) multiplied by the fixed member size. After this, the variable is ready to be used in the shader. These computations can now independently happen on a larger scale on each GPU thread. Finally, if the variable was written to, the texture is updated with information about the validity and used limbs and written to at the same index. For fixed-size integers, the process is similar. As was the case for the dynamically sized helpers, there is no need to pass information about allocated size, as it is known. The only difference is that the passed device memory needs to be copied into local thread memory. The MAP library offers simple and convenient functions to copy integer data between thread and device memory. As for CUDA, the process is nearly identical. The main difference is that CUDA does not need specific buffers and does not utilize textures to pass meta-information about the values. This makes for a much less convoluted kernel. A simple example is given below.

Code listing 4.7 Example CUDA kernel MAP library argument passing.

```
__global__ void example_function(uint32_t parallel_runs,
                             map_digit * a_used,
                             map_word * a_numbers,
                             map_digit * a_used)
{
    map_int a{a_current_length, a_used[index], &a_numbers[index * (a_current_length)]};

    func(a, ...);

    a_used[index] = a.used;
}
```

It should be noted that the library does not enforce this approach or usage of holder classes or textures. This pattern commonly appears in the code accompanying this thesis, but there is no restriction to use an alternative method of creating and using `map_int`s.

4.10.1 Random number generation on Apple Metal

The Metal shading language does not have built-in random number generation support. Since a large portion of input parameters is chosen randomly, finding a source for these numbers was essential. The selected approach was to utilize *Loki Random Number Generator* [46] repository, which contains an existing MSL library that provides an implementation based on the *Efficient pseudo-random number generation for monte-carlo simulations using graphic processors* paper [47]. This library was used through the MSL code to provide a source for generating random numbers.

4.11 Summary

This chapter describes the MAP library's scope, structure, and implementation concepts. It went over the created arbitrary-precision integer library and covered the interface, how to use the library and the contract it provides to the user. While the library is intended as a GPU library primarily aimed at Metal, additional CUDA and CPU functionality has been provided. The library is not contained to arithmetic operations but also provides a means to store and manipulate larger groups of integers easily.

Chapter 5

Paralellization using Metal API

It should come as no surprise that the metal implementation requires a significantly different approach when compared to the CPU versions. As previously mentioned, the GMP or LibTomMath libraries are unavailable for Metal API, which requires finding or creating an alternative library. This led to the creation of the MAP Library, which was introduced in the previous chapter. While this library can not compare to GMP, it allows the functional implementations of Pollard's Rho and Lenstra's elliptic curves algorithm on Apple Metal, which are discussed in this chapter. This chapter outlines the implementation of the algorithms for Apple Metal. It starts with a high-level design and structure of the code, followed by descriptions of the individual implementations, and finishes notes on the performance and optimizations of the solution.

5.1 High-level principle of the Metal implementations

In the OpenMP version, each running instance is entirely independent of the other instances. This is trivially specified in OpenMP constructs in the code. Beyond a shared flag signaling that a solution was found by one of the threads, there is no coordination or synchronization. In the case of Metal, the computation happens for a number of parallel instances organized in a one-dimensional Metal grid, where each index represents one independent running instance of the algorithm. In the Metal implementation, the computation is synchronized in stages of the algorithm (at least from the perspective of a single CPU thread). The GPU kernels, handling many parallel instances across various stages of the algorithm, are scheduled and executed sequentially. This is explained in greater detail in sections describing the implementation of the specific algorithms, but the general stage split is in initialization, main loop iterations, and loop finalization. Each stage is submitted to the GPU, and it is the responsibility of the API to schedule the work in the correct order. The host only awaits the completion of the submitted work after it schedules the final stage of the algorithm. After completion of all of the scheduled work by the GPU, it evaluates whenever a result has been found. This is repeated until a solution has been found by one of the running instances or the maximum number of attempts is exhausted.

5.2 Achieving scalability for Apple Metal

Initially, the implementation utilized a single CPU thread to schedule work on the GPU. This, however, limited the implementation in a number of ways. It limited the maximum number of parallel instances due to maximal texture sizes used in holder classes, lacked scalability, and under-utilized hardware. Splitting the work across multiple independent CPU threads allowed

for increasing the number of parallel instances but required a significant refactoring of the existing code and additional synchronization. This refactor should also allow the application to scale across different hardware and allow utilization of higher-end GPUs capable of concurrently running a larger number of computations.

The reworked implementation considers multiple independent CPU threads, each working with a separate batch (grid) of instances. The individual threads terminate as soon as they discover other threads have found a solution. This required extensive refactoring of the factorization classes to support thread-safe, multi-threaded usage. One implementation note is that this mechanism does not rely on the otherwise used OpenMP due to compatibility issues with the Clang compiler provided by Apple, which was used to compile the Metal application.

5.3 Code layout and usage of algorithms on Metal

While no class would aggregate individual implemented versions in the CPU version, as independent functions were sufficient, the larger number of shared Metal objects made introducing one very practical. The two implemented factorization algorithms, Pollard's Rho and Lenstra's Elliptic Curves, are represented by two similar classes. Those are the `MetalLenstraMT` and `MetalPRhoMT` classes. They both require a Metal Device class as their only constructor parameter and are bound to this device upon creation. This is the device on which all GPU computations for the class will be scheduled.

Once instantiated, the class has a straightforward public interface, in which a simple method called `Factorize` is available (`FactorizeETE` for *ETE* ECM). These methods require a simple set of inputs. The most important is an `map_int` parameter containing the composite number for factorization, followed by an in-out `map_int` parameter to which the resulting factor will be written. An additional group of parameters defines how the computation should be scheduled and submitted. Among those are parameters specifying the number of parallel instances, GPU threads that should be run, and the number of CPU threads that should submit the workloads. The instances are evenly split across the number of CPU threads specified. The number of parallel instances, in combination with the number of CPU threads, will significantly impact the performance of the implementation. Choosing a value too small will cause the GPU to be underused, as most of the hardware will sit idle. A value too large (such as a value above the maximum number of possible concurrently running threads) will result in the inner loop iterations taking longer, potentially losing the benefits of the parallelization.

An additional parameter allows selecting the strategy for initialization before the computation starts, such as the elliptic curve generation method or choice of a in Pollard's Rho. This resembles the strategies observed in various CPU-based versions, only now the design is more modular.

This interface specifies a parameter for selecting the desired version of the algorithm implementation, which defines how the computation should be scheduled and executed on the GPU. As the instance initialization is separate from the versioned algorithm implementation, the Metal implementation allows for combining different parameter initialization strategies with different approaches toward parallelization. This approach is more flexible than the OpenMP implementation, allowing for greater modularity if needed. However, this design approach was not overly utilized as the Metal implementations keep to simpler initialization (such as naive elliptic curve generation) and scheduling strategies, and fewer versions have been created compared to OpenMP implementation. If a need arises, this allows for the easy extension of the classes with new, more sophisticated initialization strategies.

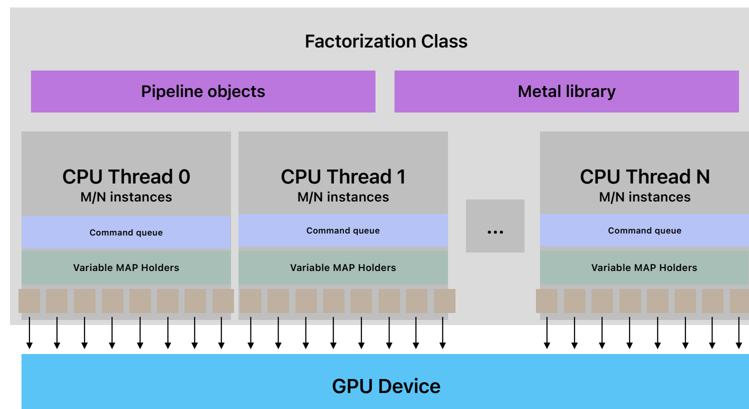

Figure 5.1 The layout of a factorization class for Metal. The diagram shows the distribution of individual instances across CPU threads and their submission to the GPU device given N CPU threads, each executing eight parallel instances.

While the algorithm classes have much in common, this similarity is not enforced by any abstraction in the code. The classes store all Metal objects required to run the algorithm; this includes MTL Functions, Errors, Argument Encoders, and Pipeline states. The example below shows how to instantiate and call Pollard's Rho factorization.

Code listing 5.1 High-level Metal factorization class creation and usage.

```
MetalPRhoMT metal_prho{device};
metal_prho.Factorize(res,
                  n_fac,
                  parallel_runs,
                  gpu_threads,
                  max_attempts,
                  strategy,
                  version,
                  cpu_threads);
```

During factorization, each class sets up `MetalArbitraryPrecisionHolder` holder per each computational or helper variable needed during the computation. As was mentioned earlier, this class holds a bulk of `map_ints` in Metal buffers, one for each instance. It then initializes the variables in parallel on the GPU according to the passed argument selection strategy. To better understand the layout of the factorization classes, look at the 5.1 figure. Both algorithms then execute their inner loops, where the computation is done in parallel on the GPU within each iteration. After an iteration is finished, it's evaluated whenever any of the CPU threads report a solution. If so, the algorithm terminates and propagates the result. If not, it continues until the maximum number of iterations specified is reached.

5.4 Lenstra implementation on Metal

This section describes the implementation of the Lenstra factorization algorithm for Metal API in greater detail. First, the CPU portion of the code is covered, and the associated Metal shaders are discussed later. The Lenstra implementation for Metal is contained within the `MetalLenstraMT` factorization class. This class has a straightforward public interface but is more complicated

in its private section. The entry point for factorization is the `Factorize` or `FactorizeETE` method, which, given the parameters, calls a low-level private versioned factorization method. For Lenstra, four distinct versions are implemented.

- *V1* - Uses EC in Weierstrass form. This version encodes EC point additions and doubling within each loop iteration of the kP multiplication into a single Command buffer. Once the inner loop is finished, an additional command buffer is committed for copying and evaluating the results.
- *V2* - Uses EC in Weierstrass form. This version attempts to reduce the overhead of using multiple Command buffers for the kP multiplications. The computation initialization and all operations within the inner kP loop, copies of results, and evaluation are encoded within a *single* command buffer. This may reduce overhead but may behave unexpectedly if k is too large, as no work is executed until the buffer is committed and the number of commands to encode grows with k.
- `ete_v1` - Same layout as Weierstrass *V1* adapted for *ETE*.
- `ete_v2` - Same layout as Weierstrass *V2* adapted for *ETE*.
- `ete_v3` - This variant has the same layout as `ete_v1`, but utilizes fixed size `map_int_f` integers in GPU shaders.

The low-level factorization methods hold the necessary computational variables in holder classes. The methods also handle additional Metal buffers for additional parameters required by used kernel functions. The methods first enter a loop that starts new rounds of ECM computations. This loop continues until it reaches maximum attempts (one attempt corresponds to one generated batch of EC curves) or until a solution is found. Each attempt first initializes the elliptic curves and points P and Q. Each instance has its elliptic curve and points. This is done as specified by the initialization strategy passed to the method. Currently, only one *naive* parameter initialization strategy is available, which picks a (for Weierstrass) and p randomly and verifies if an elliptic curve can be formed given the parameters and regenerates the values if needed. The inner loop performs point multiplication kP, with K determining the effort. This is implemented by computing repeated squaring. The value of K is doubled after kP is computed. This multiplication applies EC point additions and doubling based on the value of k in each iteration. These operations are passed to the GPU and executed in parallel. Once k is zero, the loop finishes, the result is copied to persistent memory for further possible computations, and each instance evaluates whenever a result is found. The implementation allows the computation to continue by specifying the upper K boundary. Once the boundary is reached, another attempt with new instances is made. This is the same concept as in the CPU versions. The pseudo-code in 5.2 illustrates the computation described above.

It should be noted that there may be no need to use the complicated parametrization needed to run multiple attempts with increasing size of k. The user can specify only one attempt with matching starting K and upper boundary for K. This simple parametrization would generate a single set of EC curves, compute kP and terminate.

Code listing 5.2 Pseudo-code of the implemented ECM factorization logic.

```
while (current_attempts < max_attempts && !solution)
{
    // In parallel, initialize all EC variables
    InitializeInstances(P); // Randomly chosen P for every instance
    K = starting_effort;
    while(!solution && K < effort_boundary)
    {
        // Set up kP computation in parallel
        Q = (0, 1, 0, 1) // Identity (ETE)

        k = primes_product(K);
        while (k != 0)
        {
            if (k % 2 == 1)
            {
                // Add points in parallel
                Q = AddPoints(P, Q);
            }
            // Double points in parallel
            P = DoublePoints(P, P);
            k = k >> 1;
        }
         // Computes GCD in parallel to determine if a factor was found
        FinalizeLoop(Q)
        P = Q;
        K *= 2;
    }
    current_attempts++;
}
```

The GPU portion of the code defines the following public functions that are directly called from the CPU section of the implementation. They are usually implemented in two versions, one for Weierstrass and one for ETE. Each function described in this section runs for each index in a one-dimensional grid in parallel.

- `initialize_curves` - Loops over calls to the `calculate_elliptic_curve` function until a valid elliptic curve is created or the upper limit is reached. If the return value of `calculate_elliptic_curve` is false, passed a and p is regenerated before the next iteration. This results in all instances having their own elliptic curves or being marked invalid.
- `initialize_curves_ete` - Is the equivalent for *ETE* version, it keeps a static but modifies point p to generate curves and relies on `calculate_elliptic_curve_ete`.
- `lenstra_add_points` and `lenstra_add_points_ete` - Handle point addition on EC for their respective algorithm version. It is also implemented for fixed-size integers in the ETE variant.
- `lenstra_double_points` and `lenstra_double_points_ete` - Handle point doubling on EC for their respective algorithm version. It is also implemented for fixed-size integers in the ETE variant.
- `finalize_loop` and `finalize_loop_ete` - These are responsible for determining if any of the instances found a result. It is also implemented for fixed-size integers in the ETE variant.

The class initializes the Metal functions and Pipeline States in its constructor. This action can potentially produce a non-trivial overhead. Hence, the time between initialization and the first result being returned may be slightly inflated with the time for setup that will not be present on repeated factorization calls.

5.5 Pollard's Rho implementation on Metal

This section describes the implementation of Pollard's Rho factorization algorithm for Metal API in greater detail. The Pollard's Rho implementation is contained within the `MetalPRhoMT` class bound to a specific GPU device. As with `MetalLenstraMT`, the class has a public `Factorize` method with the same parameters allowing to specify the number of instances, GPU, and CPU threads and control the maximum number of attempts. The versions are:

- *V1* - The first version schedules a single kernel using a single Command buffer in a loop. The MSL kernel differs from the *V2*. The kernel is monolithic. It performs all necessary computations for one step of Pollard's Rho and verifies if the conditions for finding a result have been met. If so, it atomically writes the index of a thread within the grid that found the result to a buffer.
- *V2* - The second version splits the monolithic kernel into smaller distinct calls for the computation of $g()$ with different parameters. There is also a loop finalizer kernel that verifies if any of the threads found a result. If so, it performs the same atomic writes with thread grid index as to where the result can be found.
- *V3* - Finally, the last version turns towards even more granular kernels, where each arithmetic operation is performed in one kernel call. This was done in hopes of eliminating register spillover that comes with more complex shaders and significantly impacts performance.
- *V4* - The fourth version adapts the *V2* approach to utilize fixed size integers.

Compared to `MetalLenstraMT`, the versions do not differ in how Command buffers are treated and committed in the code but rather in which MSL kernels are called. Either one monolithic function or a chained series of simpler functions is used for more granular parts of the computation. This variation has been considered to try to find a balance between any overheads associated with launching a new kernel and kernel complexity, which can increase the register pressure. Pollard's Rho implementation is much simpler than the ECM implementation. The following pseudo-code gives an illustration of the factorization.

Code listing 5.3 Pseudo-code of the implemented Pollard's Rho factorization logic.

```
while (current_attempts < boundary && !solution)
{
    // Each call to g() represents computation in parallel
    X = g(X, a, n);
    H = g(Y, a, n);
    Y = g(H, a, n);
    // gcd() is computed in parallel for each instance
    d = gcd(|X - Y|, N);
    if (d != 1)
    {
        solution = d
    }
    current_attempts++;
}
```

As a reminder, the function $g(x, a, n)$ is computed as $x^2 + a \ mod \ n$. As with the Lesntra implementation, on instantiation, the class initializes all Metal functions and pipelines necessary for the computation. Pollard's Rho implementation is significantly shorter and simpler when compared to the ECM implementation. The code contains significantly fewer operations, fewer divergent branches, and a significantly smaller memory footprint. The implemented MSL kernels are listed below:

- `pollards_g` - This function is responsible for calculating $g(x, a) = x^2 + a \mod N$ and writing the results. Also implemented for fixed-size integers.

- `pollards_loop_finalize` - This function is called after every iteration of the algorithm; it computes the *gcd* of the difference between x and y in absolute value. It then verifies if any of the threads found a viable solution and atomically writes the threads index into the shared variable. Also implemented for fixed-size integers.

- `prho_inner_loop` - This function is utilized in the *V1* version of the implementation. It performs all necessary computations, such as multiple computations of g and loop finalization, to complete one iteration in the algorithm within one kernel call.

- Additional kernels are a group of wrapper functions around operations of the MAP library necessary for the *V3*. They are not covered individually and are only responsible for gathering the required parameters, instantiating local `map_int` variables, calling the specific operation (addition, multiplication, or modular reduction), and writing the results.

5.6 Performance optimizations

Due to the nature of the code and some of the optimizations being inter-weaved during development, the exact measured impact, such as by how much speedup was achieved by individual optimizations for various inputs, is limited and not provided for all mentioned modifications. This is due to the inflexibility (and as a consequence of the considerable complexity this would introduce) of the code to measure the effect of each optimization independently across multiple algorithms and parameters. The impact may vary, either positively or negatively, for various inputs. Different composite numbers, amount of parallel runs, and CPU threads may show larger or smaller differences or indicate a negative direction. Consequently, the impact of some modifications on the implementation cannot be fully estimated without considering many input combinations. For simplicity, the observations and measurements presented in this section are focused on *5,052,163,649,973,526,983,733* composite, a 73-bit number which poses a sufficient challenge to evaluate performance impact while also being reasonably difficult to factor as to shorten the needed data collections across various input and variants.

Additionally, this is being considered in an unstable code-base that was significantly refactored multiple times. For more significant changes, separate, versioned implementations have been introduced. In general, the observed implementations are heavily memory-dependent. An example can be seen in figure 5.3 showing instruction costs for EC point doubling for fixed-size integers. In dynamic-sized variants, the memory wait time can even reach a staggering 60%. The biggest bottleneck is reading and writing from device memory and register spillover in the Metal kernels. It appears consistently across both algorithms in all its versions. For example, the point doubling in ETE ECM requires, in total, eleven different `map_ints` for each thread. That is, despite the variables being heavily reused in various stages of the computation. Assuming a 32-bit MAP library, each number will require *at least* 12 bytes of local thread memory and additional device memory, depending on the integer size. Considering 192-bit integers, this is an extra 24 bytes per variable in device memory, which needs to be read and written to frequently multiple times. This can quickly add up with hundreds or thousands of considered threads and is heavily visible when profiling the applications.

One possible optimization, which was considered but eventually not implemented, was shader pre-compilation. Pre-compiling the Metal shaders could save additional time, but this would likely not yield considerable benefits. Profiling showed that the time is insignificant compared to the overall run time and would likely not yield much noticeable benefit, especially when larger factored numbers are considered. Finally, for compiler optimization, the implementation utilized `-Ofast` for CPU code and `-O2` for MSL (the highest possible setting for Metal).

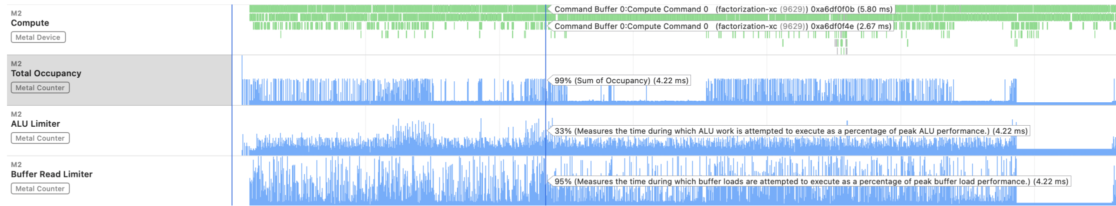

Figure 5.2 Profiling tool output for fixed size 16-bit ETE ECM variant.

5.6.1 Kernel modularity and interface

The MSL kernels mostly hit two bottlenecks. One is the very slow computation of modular reduction, which takes up a significant portion of the total computation time and is generally a costly operation. The second major bottleneck is the amount of memory and register usage dedicated to the computational variables. The second is especially apparent in ECM, which requires a larger amount of input over Pollard's Rho. One of the most crucial performance-impacting factors is the register spillover of shaders. This directly relates to the number of registers needed to successfully run and complete a shader. During spillover, the registers required by the shaders are greater than the number available.

In consequence, the insufficient number of available registers has to be compensated for by utilizing the much higher-latency memory, introducing a significant performance decrease. One of the optimizations was to make the kernel code less flexible and modular from a software engineering perspective. This was done by reducing the number of input parameters. Initially, the shaders followed a convention of the MAP library, which defined an output parameter `rop` and function signature of `f(rop, a, b)`, which effectively meant $rop = f(a, b)$. In the algorithms, the usage was such that the return parameter matched one of the inputs as $a = f(a, b)$. Simplifying the kernels and removing the `rop` value resulted in directly writing to a and fewer required registers and a noticeable performance improvement. This, however, made the kernel less reusable and generic.

5.6.2 Kernel complexity

The initial approach was to run a larger portion of the algorithm's code within one kernel to minimize the overheads of launching many kernels. However, for both algorithms, this increased complexity significantly contributed to spillover from GPU registers to main memory. For this reason, additional approaches with more granular kernels were considered. This granularity can be easily seen in the different versions of Pollard's Rho, which attempted to go from one monolithic kernel to the granularity of a single arithmetic operation. Initially, ECM utilized a single complex kernel to handle both EC point addition and doubling. Splitting this kernel from one to two distinct kernels greatly improved performance. Reducing kernel complexity reduced the pressure on registers, and an additional observed benefit was reduced L1 cache miss rate, which on some occasions dropped to one-third, a very desirable outcome. For ECM, the benefits were so clear that the original monolithic kernel approach was fully removed in all versions. Unfortunately, spillover was not entirely eliminated.

It should be noted that further granularity, with further reduction in spillover, does not necessarily yield benefits as there is additional overhead from scheduling more kernels or, depending on the implementation, additional copies from and to global memory. One interesting observation is that the EC point addition is much more costly than doubling (which uses just one EC point and, therefore, has a much smaller memory footprint). During profiling, point addition was consistently limited by buffer reads during startup, much more significantly than the doubling operation.

Runtime Shader Instruction Costs	
Wait	39,594 %
Wait Memory	39,594 %
ALU	33,208 %
Integer	25,557 %
Other	5,556 %
Shift	1,192 %
Boolean	0,903 %
SIMD Reduce	0,001 %
Control Flow	17,167 %
Direct Branch	7,825 %
Memory	10,032 %
Other	9,740 %
Load	0,140 %
Sample	0,071 %
Store	0,065 %
Pixel Write	0,016 %

Figure 5.3 Runtime shader costs for 16-bit fixed size point doubling in ETE ECM.

5.6.3 Memory layout

Focusing on memory layout and caches, the way variables are stored in the map holders unsurprisingly played a significant role in the performance of the code. The initial ECM implementation naively relied on a layout in which the same variables (mainly EC point coordinates) were stored close together, such as $AAABBB$ for three parallel instances where each needs A and B. This pattern was slightly easier to develop and use but had a significant performance cost, significantly hurting data locality. Changing this pattern to $ABABAB$ unsurprisingly improves the performance as variables accessed by a thread executing a kernel are closer together. In some instances, the measured kernel runtime is more than halved. This is a very significant but almost embarrassing optimization. An attempt could be made to provide full-stride storage of all considered variables needed for one thread. Such as by holding all needed variables in a single holder class in order as they are needed by the kernels, for example: $P_{x_0}, P_{y_0}, P_{z_0}, Q_{x_0}, Q_{y_0}, Q_{z_0}, a_0, P_{x_1}, \ldots$. In ECM, this was not explored and was limited to individual EC points.

5.6.4 Utilizing 16-bit GPU registers

Further, in hopes of reducing the register pressure, the implementation of the underlying MAP library, as well as the algorithms, were changed to be more generic and allow for the use of flexible limb size. In practice, this is limited to 32-bit and 16-bit limbs and can be specified during compilation. This allows adjusting the size for the specific targeted platform and hopefully improves performance. Since Apple GPUs utilize 16-bit registers, and using smaller data types is encouraged [48], this was the size chosen for Metal implementation. The flexibility of the compile-time switch allowed straightforward measurements to evaluate the usefulness of this optimization. While spillover was reduced across kernels, this did not yield the desired benefits, and the amount of time the shaders were waiting for memory has surprisingly increased.

It should be noted that using 16-bit limbs may reduce overall data usage, but this will also nearly double the iterations needed for certain operations, which could outweigh the reduced spillover. Unfortunately, while the 16-bit variants fared decently well, no significant positive impact was observed on Metal. An interesting note is that while this outcome was disappointing and unexpected for Metal, especially given the effort needed to refactor the MAP library and the majority of the algorithm code, the effect of this was different on CUDA, which will be discussed in later sections.

5.6.5 Utilizing fixed-size large integers

As discussed earlier, there are technical difficulties in using dynamically sized arrays in thread memory that disappear when fixed compile-time sized `map_int_f` is used. This move can potentially help the compiler optimize the code, such as with loop unrolling. An additional impact of using fixed-size `map_int_f` may be with the switch from device to thread address space. Ideally, the kernel would load the data from slower device memory to faster, closer memory, perform a set of operations, and, upon completion, write the results back. When observing this implementation, this change heavily increases memory spillover over dynamic versions. This suggests that the used arrays spill out and that this scenario, unfortunately, did not occur.

Similarly to CUDA, certain conditions, such as dynamic indices, prevent arrays from being stored in registers. [42] This likely means that the data remains in higher-latency memory, and any performance benefits are due to other compiler optimizations. Previously, a single integer variable required 12 bytes of thread memory and additional data in device address space. Now, for a 192-bit integer, it is 8+24, slightly reducing the number of registers needed to store the structs. All available MAP library functions for the GPU had to be adopted to use `map_int_f` to implement this change. While this didn't need any algorithmic changes to the code, it nearly doubled the size of the library. Unfortunately, as with the 16/32-bit split, this optimization did not deliver any significant improvement for the Metal implementation, but as was the case for limb size, it was much more noticeable on CUDA. Given that this was not a successful endeavor, one additional option could be considered. Keep the integer data in the device address space and reduce the size of the structure by one-third by removing the length.

5.6.6 Modular reductions to reduce memory bottlenecks

One observation was the introduction of additional modular reductions to EC point addition and doubling noticeably improved performance for observed numbers. This effect manifests in between two bottlenecks. One is the memory usage, while the other is the number of computational operations. Larger number arithmetic takes longer, uses more memory, is less efficient with cache, and is more likely to lead to register spillover. The other point is that modular reduction is an incredibly costly operation, but with the result being a possibly much smaller number, using it can bring benefits, such as if the result will be used in additional arithmetic operations. Those arithmetic operations will need to perform fewer steps and require reading and writing less data. To provide a simple example, assuming 16-bit limbs, when squared, the previously mentioned 73-bit number results in a 145-bit integer. This can increase the amount of accessed limbs from 5 to 9. In some observed cases, additional modular reductions could double the performance of the implementation. The observed differences in the mean runtime can be seen in the 5.5 figure. This only emphasizes the significance of the observed memory bottleneck. Even increasing computational complexity in hopes of reducing memory footprint was worthwhile. Introducing additional modular reductions to the code reduces the memory footprint during the computation and permits allocating less memory beforehand as the numbers appearing during stages of the computation will be smaller. This, however, should not be considered a universal case applicable to the factorization problem. Instead, it should be regarded as a step to address one of the dominating bottlenecks in this implemented solution and its constraints. This effect likely manifests itself because of inefficiencies in the solution. The MAP library modular reduction is implemented inefficiently, and further effort into optimizing it would likely yield considerable benefits. In addition, the MAP library could be extended to include optimized operations such as modular exponentiation.

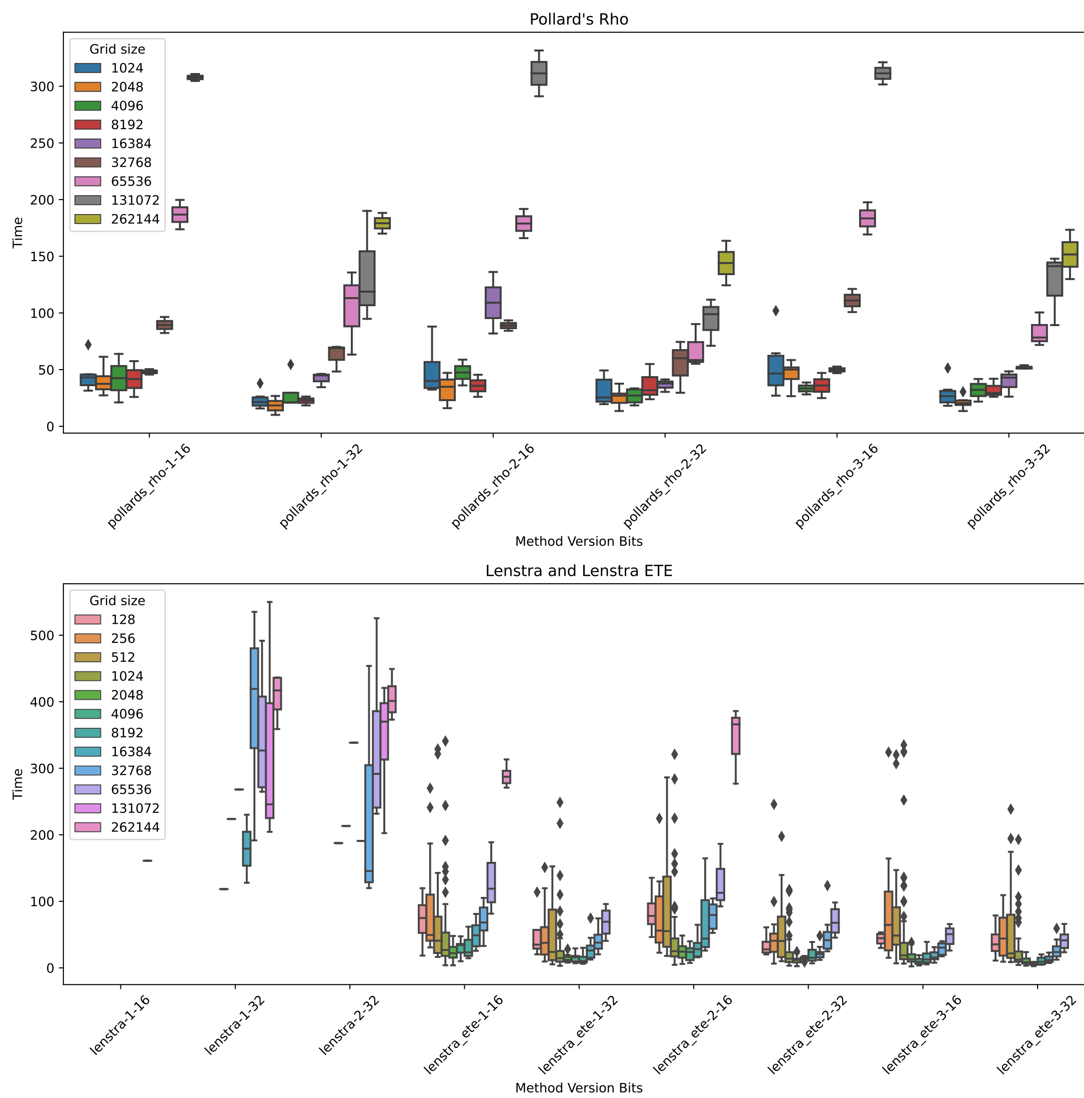

Figure 5.4 Aggregated meantime for individual Metal variants. Capped at upper time limit.

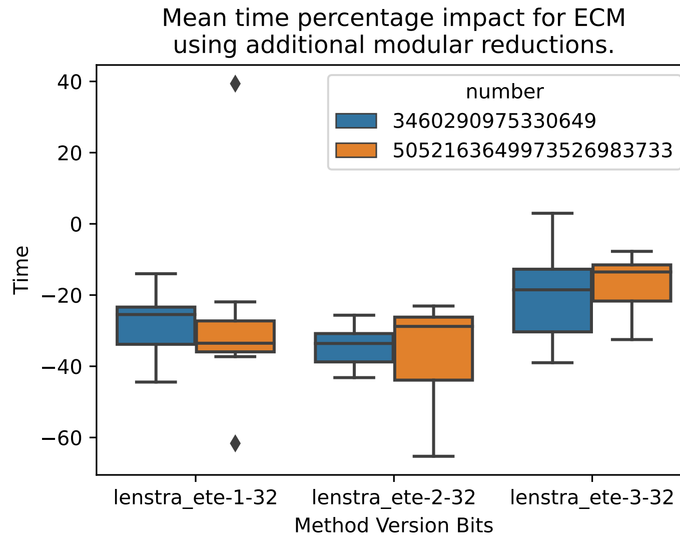

Figure 5.5 Percentage impact of additional modular reductions on the meantime.

5.6.7 Version and parameter performance impact

Given the extensive combination of possible parameters affecting performance and randomness in the algorithms, determining which combinations are well-performing and which give unsatisfactory performance was challenging. The results are hardware-specific, but observing and summarizing those will enable a more fair comparison of individual implementations described in earlier sections. A sufficiently large data sample is needed to evaluate the impact of different parameter combinations. For each number to be factored, each algorithm and each version, various combinations of algorithm inputs, memory reserve for computational variables, number of CPU threads, and number of instances per thread must be repeatedly measured to address randomness. Discarding intermediate measurements during development, the full comparable dataset contains ten thousand measurements. To properly evaluate the results, it was necessary to reduce the impact of "one-time" lucky results. This is necessary so that non-optimal parameter choices are not considered among the best. This was achieved by considering only combinations that successfully factored the composite number in most measurements and had to be measured multiple times. The mean and median factoring time was taken across these measurements to evaluate the best consistently performing parameter combinations. The conclusions were then drawn from an upper sample of these candidate measurements.

Starting with CPU threads, the most successful choice has been 1 and 2 threads for both algorithms. In the Metal implementation, there is a direct relationship between CPU threads and grid size, as the texture widths limit the grid size associated with a single thread. In practice, this sets a maximum of 4,096 curves per thread for ECM or 16,384 instances in Pollard's Rho. Looking at the impact of various grid sizes based on measurements and profiling, it seems that for the considered hardware, there is an upper limit somewhere between 8,192 and 16,384 instances, after which the performance starts to degrade sharply. Figure 5.2 shows occupancy and selected limiters with 12,288 curves, showing high saturation. The fixed-size variants seemed to fare better over dynamic-sized. Overall, the best results were offered by *V3* ETE version. For both algorithms, the best results could be seen with 2,048-4,096 grid sizes, 1 or 2 CPU threads, and 32-bit limb size. With block threads, the observed differences were slight, with 32 and 64 showing the best results. Sizes smaller than 32 likely have no effect, and the API sets the size to 32. In contrast, for larger sizes, there is an upper limit for individual shaders, and the

dimensions are automatically capped at the limit. Figure 5.4 shows a comparison of aggregated results. Observing the different versions, it seems that the various methods of command buffer submission did not have any significant impact on performance.

5.7 Summary

In this chapter, the structure and design of the algorithms for Apple Metal was described. The differences and evolution of the implementation through various versions were outlined. This was eventually finalized with notable attempted optimizations and their impact (or the lack of) on the solution's performance. Both algorithms have been successfully implemented for Apple Metal utilizing the created MAP library. The implementations heavily suffered from memory bottlenecks, which were, unfortunately, not overcome. While differences in configurations have a significant impact on performance, the same cannot be said about individual versions of the algorithms. This can largely be attributed to the same observed bottleneck, which overshadowed other differences. By a narrow margin, the fixed-size variants were established as better than the dynamic-sized ones.

Chapter 6

Paralellization using CUDA

This chapter describes the implemented CUDA version and its differences and similarities with the Metal implementation. Given that the implementation is similar and shares many of the core concepts, the description in this chapter is less detailed and focuses more on differences. As in the Metal section, the necessary functionality, such as storing arbitrary precision numbers and the underlying support for arithmetic and algorithmic operations, is discussed. This is followed by a description of the implementation of the individual algorithms built on top of the supporting code. Finally, the implementations are observed from a performance-oriented perspective.

6.1 Arbitrary integer precision arithmetic on CUDA

As was discussed in previous chapters, there are several libraries for arbitrary precision arithmetic on CUDA API, such as *CGBN*, *CAMPARY*, or those created within various theses. These frequently seem to focus on paralleling the arithmetic operations themselves, and some, primarily the latter, seem to lack support for more complex operations such as finding GCD, modular reduction, or even division. While speeding up the individual computations of a single instance of a factorization algorithm would undoubtedly be beneficial, this thesis attempts to run many concurrent instances of the algorithm in parallel without thread cooperation, hoping to achieve speedup with a greater number of instances and random starting points.

6.2 Storing arbitrary precision integers

The arbitrary precision integer representation and storage relies on the `map_int` CPU-based definition. The CPU definition differs from the Metal GPU definition, as can be seen in the Metal API implementation section, by lacking a pointer address space specification. The CUDA API is more flexible than the Metal API, allowing the CPU and GPU code to share the struct definitions. Reusing the `map_int` structure enables the implementations to share CPU functionality without modification. What is more significant is that CUDA does not require distinct address spaces to be defined. This theoretically allows an implementation in which various instances can utilize different memory without the need to adjust the implementation of the library or dynamic allocation. This offers an additional path for potential optimization, where heavily used variables can be preferentially placed in faster memory (such as in on-chip shared). In contrast, lesser utilized variables can remain slower in memory (global, local), essentially giving more influence over potential spillover.

6.3 Extending the Metal Arbitrary Precision library

The CUDA implementation relies on the underlying `map_int` structure, so CPU-based functions of the library can be shared without modification. However, the `MetalArbitraryPrecisionHolder` class relies on Metal API-specific buffers and textures, which cannot be reused for CUDA. This required a separate implementation adapted for CUDA. This led to a newly created `CUDAArbitraryPrecisionHolder` class sharing a majority of its public interface with the Metal version. However, instead of the underlying and publicly exposed storage being formed by Metal Buffers, pointers of the specific type are exposed and can be directly passed to CUDA functions. The allocated memory is allocated through `cudaMallocManaged` and allows access to the memory for both the host and device code. Otherwise, the class offers the same functionality and serves a similar role of owning a large number of related `map_int` variables stored in a suitable format. This concludes the modifications required for utilizing the library in the host section of the code.

For the library GPU functions, the differences are numerous but straightforward. An additional, separate definition and declaration of GPU functions is necessary and requires porting from Metal Shading Language. This required creating a matching set of `device` functions and removing address space specifications. To avoid conflict and easily distinguish between individual function versions, the CUDA implementations have a prefix of `cap` for CUDA Arbitrary precision instead of `map` for Metal. The potential conflict could be problematic, as the functions could clash with the implemented CPU version.

Code listing 6.1 Comparison of MAP library CUDA and Metal variant function declarations.

```
// Metal:
bool map_copy(thread map_int & rop, thread const map_int & src);
// CUDA:
__device__ bool cap_copy(map_int & rop, const map_int & src);
```

The example above shows a simple function re-declaration from the Metal version to the CUDA version. The process of porting the library was mostly simple but tedious, requiring many but simple changes. One notable difference is in fixed functions (with `_f` suffix). Metal did not allow varying address spaces in one definition. Hence, all memory was declared in the thread address space, and there was no need to pass helper variables in fixed-size function calls, as they could be created locally, providing the same characteristics. For CUDA, this restriction is not present, and this property of the dynamically sized functions was preserved. This gives the kernel utilizing the library control over where each used variable is stored.

6.4 High-level principle of the CUDA implementations

The CUDA implementation is similar to Metal in many ways. The principles and patterns are reused and modified to apply to CUDA. The implementation is processing a large number of instances synchronously in parallel. This work can be split amongst multiple CPU threads, each handling a set of synchronous instances in a one-dimensional grid, for which work is being scheduled to the GPU. Each algorithm stage is handed off to a CUDA kernel and executed. Each CPU thread continues to submit work, possibly in different stages, as each thread works with its own CUDA stream until it reaches an upper boundary or a result is found in one of the threads. The CUDA implementation does not have the same number of instances limitation as the Metal implementation.

The code layout and structure of CUDA implementation match that of the Metal implementation, with minor adjustments. The created classes, naturally, have different internals matching the used framework, but public-facing methods are equivalent in structure and usage. As CUDA is a higher-level API over Metal, it does not require explicit encoding of parameters passed to the GPU kernel, resulting in simpler code and faster development.

Code listing 6.2 CUDA Factorization class creation and usage.

```
CUDALenstra cuda_lenstra{};
cuda_lenstra.Factorize(...);
```

6.5 Lenstra implementation on CUDA

As mentioned before, the implementation is contained within a single class, in this case, `CUDALenstra`, with an identical public interface as that of the Metal implementation (mainly the `Factorize` and `FactorizeETE` methods). With CUDA, there is no need to work with Command buffers and their submission, and as a consequence, only one version of how the work is scheduled to the GPU was implemented for both Weierstrass and ETE. The CUDA implementation offers only a single version for Weierstrass and two for ETE, for which both fixed and dynamically sized options are available. The implementation relies on lower-level factorization methods, which set up the needed resources such as `CUDAArbitraryPrecisionHolder`s, which depend on CUDA API to allocate managed memory. During the main computational phase, only the GPU uses the memory, and potential results are only accessed by the CPU at the end of the computation. This means memory transfers should not have a significant impact. Once again, the flow of the algorithm is the same as on Metal. The algorithm starts by generating all elliptic curves in a one-dimensional grid. The generation happens in parallel on the GPU using the parameter-specified method. The algorithm then enters the main computational phase, where the points P and Q are added together and doubled, again done in parallel. Once the loop is finished, the results are evaluated. If any instance finds a result, it is returned, and the computation ends. With the Weierstrass version, there is a single CPU-based GCD computation of the instance result and N, which is then returned. For ETE, there is no additional CPU computation. The GPU portion of the code is defined within a list of kernels needed for each stage of the algorithm.

- `initialize_curves_3d` - This kernel is responsible for initializing each instance for the Weierstrass implementation. Before it is called a, p_x and p_y are set to a random number. Within the kernel, it is verified if a valid elliptic curve can be formed. If not, it makes small adjustments to the previously randomly generated inputs until it succeeds (or runs out of attempts to do so). Once the kernel finishes, all instances will have elliptic curves to continue the computation.

- `initialize_curves_4d` - Symmetrically to Weierstrass, the ETE version first sets P_x and P_y randomly. These parameters are then used to verify a valid curve can be formed, making adjustments if not. Finally, the last step in the kernel is to compute P_t given P_x and P_y and set P_z to 1.

- `lenstra_add_points` and `lenstra_add_points_ete` - Handles point addition on EC for their respective algorithm version.

- `lenstra_double_points` and `lenstra_double_points_ete` - Handles point doubling on EC for their respective algorithm version.

The CUDA implementation does not need to initialize GPU kernels, pipelines, functions, encoders, or command buffer objects, as was the case for Metal.

6.6 Pollard's Rho implementation on CUDA

Pollard's Rho on CUDA has been implemented in three versions: the *V1* and *V2* match those seen in the Metal chapter, approaching the problem with varying kernel complexity, the third *V3* version structurally matches *V2*, utilizing smaller kernels adapted for fixed-size `map_int_f`. The implementation is contained within a factorization class, offering the previously discussed public interface and the ability to split the factorization across multiple CPU threads, each submitting work to an independent CUDA stream. All versions share the parameterized initialization process, which is modular but is implemented in a single variant. First, each CPU thread allocates sufficient CUDA-managed memory. Parameters x and y of each instance are set to two, and a is selected randomly. The differences in versions appear in the main computational loop and whenever it is split into multiple, simpler kernels in consecutive calls or a single larger kernel. A brief description of the individual kernels is given below:

- `pollards_rho` - Is a single cohesive kernel, offering one full algorithm iteration. It performs three calls of $g()$ and then computes the GCD to finalize the loop. If any of the GPU threads finds a result, the threads index in the grid is written to a shared variable. This kernel is associated with the *V1* implementation.
- `pollards_g` - Is a simple kernel performing one calculation of $g()$. It consists of three MAP library operations: multiplication, addition, and modular reduction. The kernel is available in two forms for dynamic and fixed-size integers and is utilized by *V2* and *V3*.
- `finalize_loop` - The loop finalization computes the difference of x and y, the GCD with n, and writes the thread index if a result was found to a shared variable. The kernel is available in two forms for dynamic and fixed-size integers and is utilized by *V2* and *V3*.

6.7 Performance optimizations

It is clear that the Metal and CUDA implementations, beyond API differences, are very similar. As such, many of the optimizations applied in Metal were carried over to the CUDA implementation. With the CUDA implementation being created after Metal, some of the mentioned optimizations were already considered from the beginning. Hence, no estimate of their impact is given. Briefly, the reduced number of input and output parameters for kernels (reduced modularity) has been applied in CUDA. The same applies to the smaller kernels in Pollard's Rho and ECM. The same variable layout relying on memory locality has been used. The most notable optimizations are the additional modular reductions and, more significantly, the utilization of fixed-size integers. The details of which are discussed in the following sections. The compilation flags used for the implementation can be seen in the listing below.

Code listing 6.3 Compilation flags used for CUDA variants.

```
-Xptxas -O3,-v --default-stream per-thread -gencode=arch=compute_89,code=sm_89
```

Note the default stream flag is set to per-thread, which is crucial for the implementation to utilize CUDA streams correctly.

6.7.1 Modular reductions to reduce memory bottlenecks in ECM

As for ETE ECM in Metal, additional modular reduction has been added to the EC point addition and doubling operations. This has reduced the stress on memory and has brought noticeable performance improvements. This has been consistently observed across measurements.

A figure 6.1 is given to visualize the performance improvements. The figure shows the relative change in mean runtime across various parameter combinations (totaling around three thousand total measurements). As was mentioned for this optimization in Metal implementation, this is far from ideal, as additional expensive operations are performed and are likely only beneficial because of the existing bottlenecks in the solution. Well-optimized operations, such as modular exponentiation, would likely yield better results.

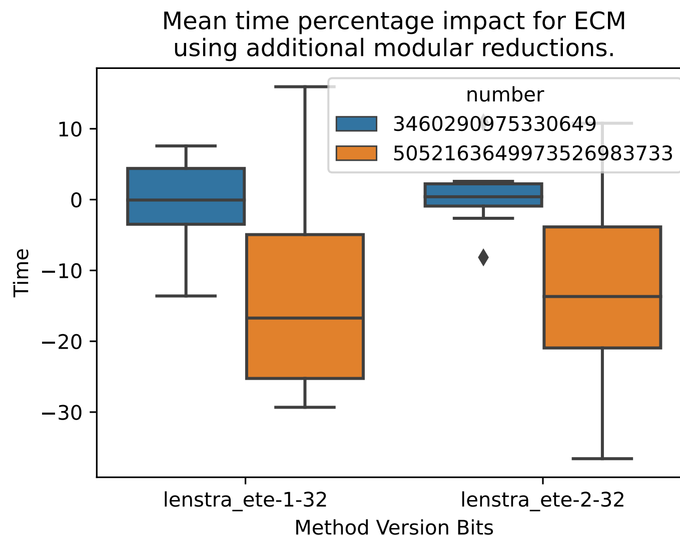

Figure 6.1 Mean time percentage change when additional modular reduction was applied.

6.7.2 Utilizing fixed-size large integers

The switch to a fixed size `map_int_f` integers has been much more successful on CUDA when compared to Metal. The effect of this change can be observed in figure 6.2, which shows aggregated results across versions. The implementation versions utilizing fixed-size integers dominate over previous versions for both Pollard's Rho and ECM. As in Metal, fixed-size arrays allow the compiler to optimize better. While a very beneficial optimization, this likely did not significantly impact memory utilization. The amount of spillover memory did not change compared to non-fixed-size functions. This is not unexpected as CUDA utilizes registers for local arrays only under particular conditions, such as when constant indices are used. [49]

6.7.3 Using shared memory for N

One attempted optimization on CUDA was to utilize shared memory in CUDA to reduce spills. This optimization has been hinted at in preceding sections, which outlined the possibility of varied address spaces. The factored number, or N as appearing in the code, is never written to and is the same across all running instances. It is accessed multiple times in the kernels and frequently appears in costly modular reductions. Moving this variable into shared memory could potentially bring benefits. Not only is the stress on local memory reduced, but it allows utilizing the much faster on-chip shared memory in critical regions of the code. The mechanism was implemented for each thread block, where the first thread copies the passed value into a shared array. All threads move on with the initialization of additional variables, after which they are all synchronized and continue with the computation utilizing shared N. Unfortunately, this attempted optimization did

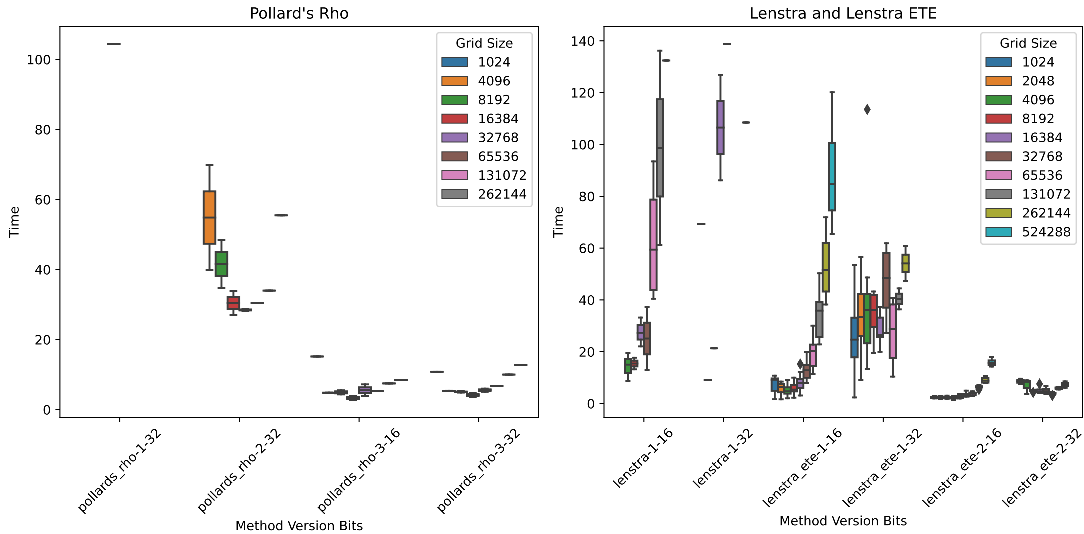

Figure 6.2 Aggregated meantime for individual CUDA variants. Capped at upper time limit.

not deliver on the expectations and significantly decreased performance. Hence, it was removed. This result did not improve even for larger thread counts sharing and reading N.

6.7.4 Using 16-bit word size

For Metal, there was a clear motivation to attempt to use 16-bit integers. It allowed more granular storage needs and was directly supported by the hardware, and it is recommended as a possible optimization path at a WWDC talk about Metal optimizations. [48] In CUDA, there was no similar preconceived motivation. Still, with the implementation being built on the same library, the property of choosing between 16 and 32-bit was also available essentially for free. The interesting aspect is that the measured difference between the 16-bit and 32-bit seemed to favor the 16-bit option. This varied across versions and was the least significant in the latest fixed-size versions of both algorithms. During closer inspection, it was revealed that the 16-bit variant used fewer registers, slightly reducing spillover and improving performance. It is worth mentioning that the choice of 16 bits does not only impact the limb size but also a large number of other variables appearing in the library operations.

6.7.5 Version and parameter performance impact

The CUDA implementation has the same number of input parameters, allowing tweaking the number of curves, the number of trials, how they are distributed across CPU threads, what effort should go into one curve trial, or how many threads should run in a block. Together with multiple versions of the implementation, this brings in many combinations to measure. Overall, it seems that the CPU thread count did not matter much. Instead, the size of the grid, which indicates the total number of parallel instances, did. The best performing sizes seemed to be 16,384 and 32,768. While the observed differences were minor, 32-thread blocks seemed optimal. There was a visible improvement when moving from smaller sizes up, but this improvement stopped at 32, from 32 onwards there is a very slight degradation. With 1,024 thread blocks, the implementation failed to produce results due to exhausting resources. This behavior can be observed for 16-bit fixed-size ECM in the 6.3 figure. For ECM, the ETE variant outperformed Weierstrass. The 16-bit variant seemed to dominate the 32-bit variant consistently across versions

for both algorithms. The fixed-size variants have consistently outperformed dynamic sizes. This was the case even when carefully grouping the comparisons to match specified parameters. This avoids wrongful conclusions when less favorable parameters would be measured more frequently for specific variants influencing the aggregated mean. The observations were the same even when comparing runtime in non-aggregated form over specific parameter combinations. However, this is a much more convoluted and difficult-to-read format; hence, most of the visualizations present show results in a more straightforward aggregated form.

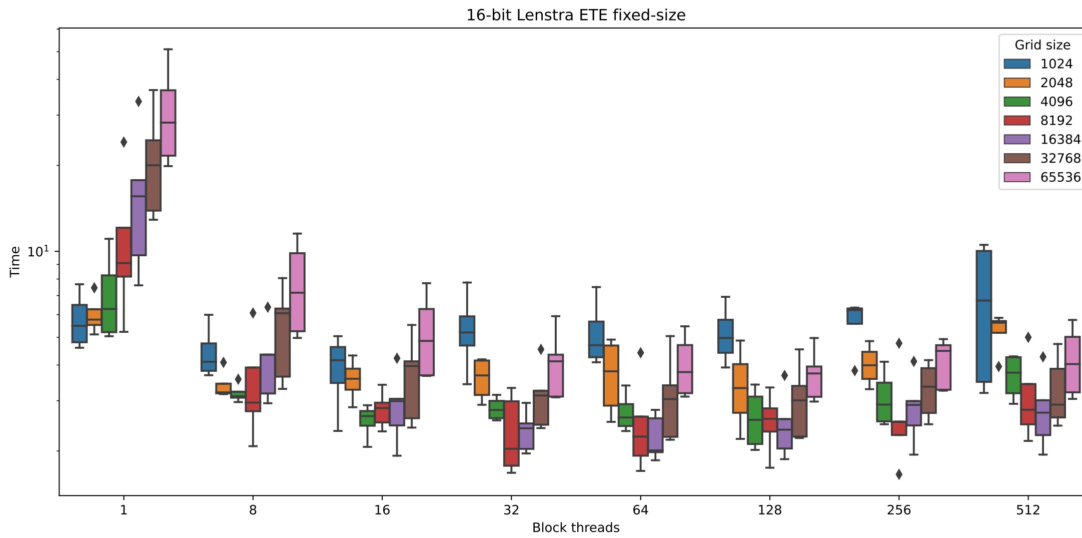

Figure 6.3 Mean time for various block thread sizes on 16-bit fixed ECM-ETE

6.8 Summary

This section described the structure of the CUDA implementation, and it emphasized the differences and similarities with Metal and how different attempts to improve performance affected the implementations. Both selected algorithms were implemented in multiple versions utilizing the ported MAP library. As in Metal, the implementation greatly suffered from spillover and memory bottlenecks. The fixed-size 16-bit variants seemed the best for both Pollard's Rho and Lenstra.

Chapter 7
Results analysis

This chapter attempts to provide a comparison across various discussed and implemented solutions. It strives to give more insight into the performance of the implementations and differences observed across different platforms and approaches. It does so using several selected composite numbers with two distinct prime factors.

7.1 Considerations for measurements

Comparing varying implementations across different platforms is challenging. As observed in earlier chapters, the inherent randomness of the results complicates the evaluation. It is difficult to separate lucky starting points from consistent performance, and the difficulty only increases when, due to the nature of the implementations, different hardware needs to be considered. Due to this nature, no striking conclusions can be made with good consciousness without careful consideration. The first step is to select reasonable metrics for comparison. One useful metric is the throughput of the implementation. This can be defined in various forms and is tied to the algorithm. It can be the amount of distinct inner loops computed for Pollard's Rho or the number of finalized trials in the case of ECM. The problem is that the implementation needs to support acquiring any throughput metrics. This is the case for most solutions and versions created in this thesis. Unfortunately, such metrics are not always easily available for other solutions. For this reason, comparison across solutions will still be dominated by runtime as a simple, consistently available, and measurable metric. Additional comparisons will be made regarding scaling and resource usage of individual solutions.

In this thesis, the implemented variants were the CPU-based sequential and OpenMP versions, followed by CUDA and Apple Metal implementations. Additionally, notable widely available implementations are considered for comparison, which will be briefly discussed in the following sections. The implementations will be compared across a variety of inputs and composite numbers, but not all implementations are considered for all input numbers. This is due to the difficulty and time requirements to gather a sufficiently large sample for evaluation. The considered composite numbers all have two distinct prime factors and are listed below:

- *3,460,290,975,330,649* - 52 bits with 20 and 32-bit factors
- *5,052,163,649,973,526,983,733* - 73 bits with 37-bit factors
- *870,729,462,492,667,946,890,471* - 80 bits with 40-bit factors
- *410,008,714,444,926,584,643,751,636,103* - 99 bits with 49 and 50-bit factors
- *740,823,820,721,940,713,928,228,049,555,961* - 110 bits with 50 and 60-bit factors

- *1,070,868,838,929,384,612,779,308,083,256,678,872,73* - 127 bits with 64-bit factors

Since CUDA and Metal can not run on the same hardware, at least two different systems need to be used for CUDA and Metal implementations. Two devices were used to measure the runtime results. The first device was utilized for Metal and CPU-based computations and came with an Apple M2 chip with 8 CPU (ARM) and GPU cores and 16 Gigabytes of unified memory. This is, unfortunately, on the lower end of available Metal-capable hardware, possibly contributing to some of the limitations encountered. The CUDA implementation has been measured on a system with an Intel Xeon E5-2620 (x86-64) CPU, 32 GB operational memory, and NVIDIA GeForce RTX 4070 Ti. When discussing specific GPU metrics, an additional NVIDIA GeForce GTX 1650 GPU will be considered; this GPU provides similar performance to the M2 and allows the collection of additional metrics that require elevated privileges on the host system.

7.2 MAP Library and GMP comparison

One of the key aspects of the performance of the individual GPU solutions is determined by the performance of the MAP library. This section shows a simple exercise in which the GMP library is compared against the MAP library for CUDA. It seeks to, at least partially, answer the question of how many integer operations one needs to perform for the CUDA parallelism and its overheads to be beneficial. This aspect could be studied in greater detail and will vary across different operations and hardware or sizes of considered numbers. This section is mainly for illustrative purposes and will be constrained to a few selected operations, one system, and number sizes up to 20 32-bit limbs. The considered operations will be a single addition, three chained additions, and a binary shift to the left. The considered operations are performance-wise on the less costly side. Those will be observed for CUDA with fixed and dynamic-sized numbers and compared to a sequentially running GMP. Each operation will be performed on distinct numbers. It is likely a safe assumption that more complex operations, such as multiplication, would show much less favorable results for the MAP library, as various GMP optimizations, such as mixed usage of multiplication algorithm, would play a bigger role.

Figure 7.1 shows a growing number of operations performed compared to execution time for CUDA and GMP to finish the total number of operations. It can be seen that launching a CUDA kernel with MAP library started to yield benefits roughly between 4,096 and 8182 operations. Interestingly, this dropped to around 1,024-2,048 operations for three chained additions. For binary shift, which is a relatively simple operation, the boundary was again around 4,096. These results show that a speedup over GMP is possible, although it requires a rather large number of operations that can be parallelized to materialize.

7.3 CUDA and Metal implementation comparison

The GPU implementations have been restrained up to the 99-bit composite, which provided a sufficient challenge for the implementations. From run time considerations, the CUDA implementation outperformed the Metal implementation. This is, however, as was mentioned observed on different hardware due to API support. The overall measurements can be seen in 7.6 figure. It can be seen that for the 52-bit input, CUDA had a somewhat higher latency. For Metal, Pollard's Rho implementation can be seen lagging behind other variants, while CUDA is keeping pace much better but is outperformed in 73 and 80-bit inputs by ECM. Considering all GPU variants, the best results were observed for ECM implementation on CUDA.

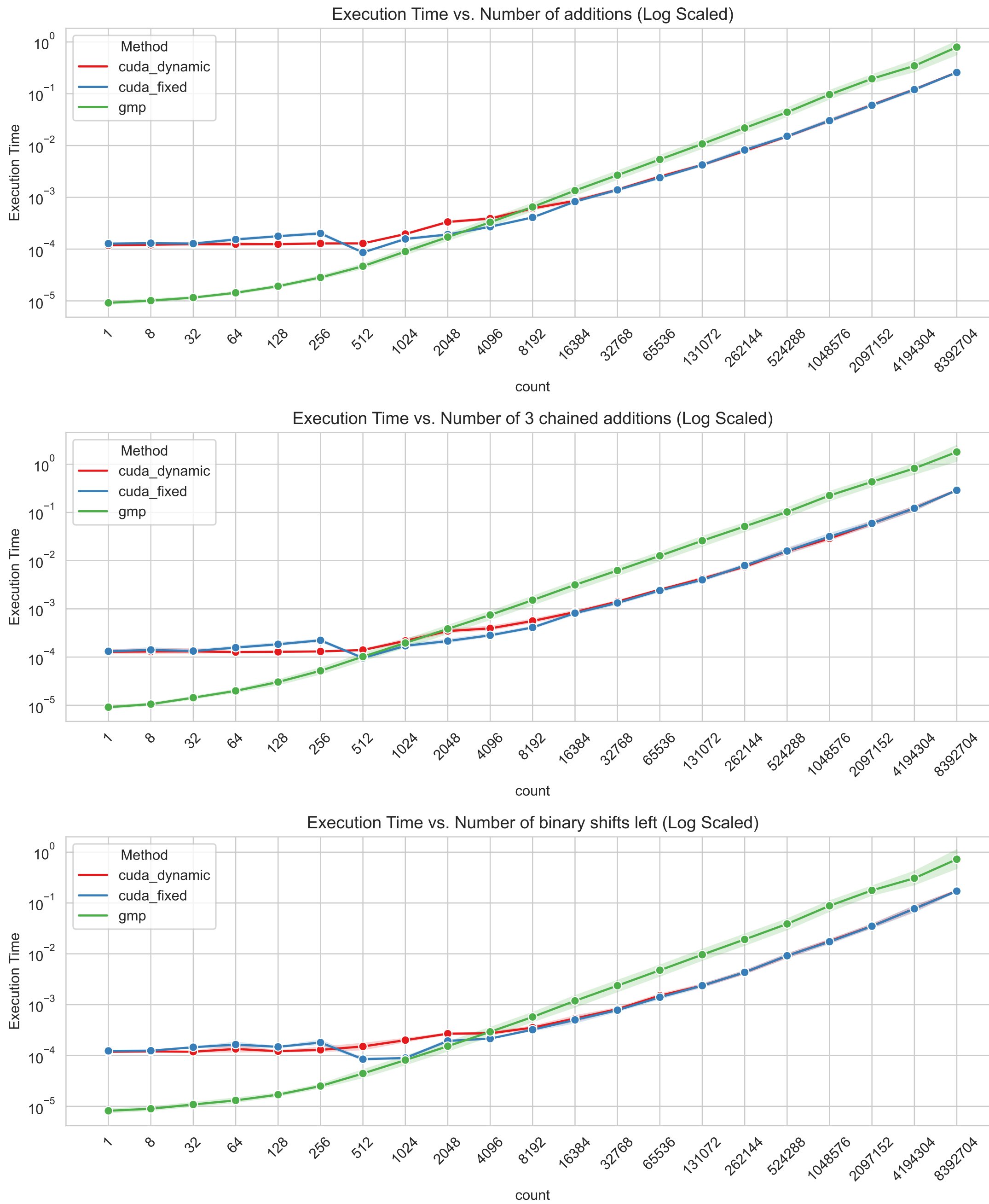

Figure 7.1 Comparing time required for sequential GMP library operations vs CUDA MAP library parallel operations.

For some measurements, the CUDA variant was run on two distinct GPUs. One was a less performant NVIDIA GeForce GTX 1650, which delivered results similar to the M2 GPU. The other was the previously mentioned higher-tier 4070 Ti, capable of handling more parallel instances, delivering the best results around 16,384 and 32,768 grid sizes. At the same time, the peak performance was substantially lower for Metal in 2,048 instances. If we recall the optimization discussion from the CUDA chapter, it can also be noted that some noteworthy optimizations have been more successful on CUDA. The CUDA implementation has been easier and faster to implement and optimize, given it is a higher-level API with less convoluted mechanisms. An additional benefit of CUDA is the widely available hardware that allows scaling far beyond what was considered in this thesis. Rather than focusing on a raw runtime comparison, the focus will be on what is perhaps a fairer comparison between the GPU implementations, which is the utilization of resources they use to produce a result. Instead of comparing the results of the CUDA and Metal implementation on matching parametrization, which considers different hardware likely to have limited informational value, the implementations were compared at various points where they displayed good performance. Figure 7.3 shows the mean execution time for various grid sizes. It can seen that the minimum meantime is further for CUDA over Metal.

First, let's observe Metal ECM *V3* on the 80-bit composite number. On the 2,048 grid size, the total occupancy is very low, around 10%. When moving toward 4,096, it nearly doubles, but it becomes apparent that the solution is starting to see issues with cache utilization. For 8192, the occupancy remains similar to 4,096 values, and again, the performance is limited by cache utilization. For a change, if we observe *V1* in 2,048, the prominent performance limiter is the buffer read and write limiter. While cache plays a significant role, it seems to be less of a concern compared to *V3*. The occupancy remained similar for both *V1* and *V3*. The most interesting observation is how cache seemed to play a more significant role for the fixed-size *V3* variant.

For *V1* Pollard's Rho on Metal, the 2,048 grid size sees similar occupancy as in ECM, with buffer reads and writes being the most significant limiter. However, for Pollard's Rho, the occupancy grows up to 8,192 instances, reaching 26%, after which it stagnates. On composite numbers of similar size, Pollard's Rho implementation can achieve higher occupancy. Interestingly, for *V3*, the occupancy at 8,192 is higher than on *V1*, reaching around 33%, but does not grow further. The fixed-sized *V4* struggled to grow beyond 20% occupancy for the 73-bit number assuming six 32-bit limbs but did reach 30% when the fixed size was reduced to 3 limbs. For the *V1* and *V3* versions, it seems that cache utilization was less of a bottleneck, with buffer read and write bottlenecks being dominant. This was not the case for *V4*, as in fixed-size ECM.

Occupancy for CUDA will be observed on the NVIDIA GTX 1650 GPU. For *V2* ECM on CUDA, considering the 80-bit composite, the occupancy at 1,024 is around 5-7%, 2,048 is similar as on Metal, around 10-13%, 4,096 sees around 30-40% and growth at 8,196 is limited with 40-45% occupancy. For grid size above 8,196, the occupancy is between 40-50% and does not grow beyond 50%. When considering *V1* occupancy grows as 5-7%, 9-13%, 25-30%, 40-50% at 1,024, 2,048, 4,096 and 8,196 respectively, after which it stagnates at 40-50%. Finally, for Pollard's Rho, the *V1* variant sees 5-7%, 10-13%, 17-23%, 25-35%, 35-45%, 45-50% for 1,024, 2,048, 4,096, 8,196 and 12,228. While for *V2* it is 5-7%, 10-13%, 20-25%, 30-40%, 38-45%, after which it struggles to grow further. For the CUDA variants, the limiter once again seems to be the cache and memory reads and writes.

Interestingly, occupancy did not seem to vary with smaller or larger composite numbers in most cases, with the exception of fixed-size ECM variants for Metal. For those, occupancy was positively or negatively impacted by a smaller or larger size of the fixed map integers. This effect was not observed for CUDA and dynamically sized variants. In the figure 7.2, it can be observed that with a growing grid size, the runtime of a kernel is growing. For example, in *V1*, the EC point doubling kernel at 1,024 instances could take around 16 ms and 60 ms for 4,096, while for 32,768, it could be around 260 ms. On Metal, the point doubling for 1,024 instances took around 11 ms but grew to 15 ms at 4,096. When observing the specific kernel (EC Point doubling is the longest running kernel for ETE ECM), it is clear that the fixed-size variants

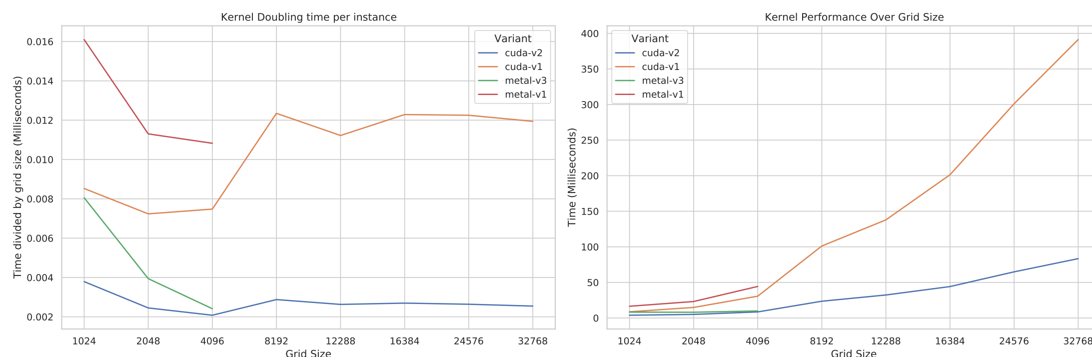

Figure 7.2 Mean runtime and single instance time for ETE EC point doubling kernels over specific grid sizes. Note that the Metal variant is limited to 4,096 due to texture width limitations.

outperformed dynamic sized on both Metal and CUDA and that overall fixed-size CUDA offered the best performance.

To summarize, the CUDA variants were able to achieve higher occupancy compared to Metal, and the growth continued further with a larger number of instances. This is aligned with the observations in the 7.3 figure, where the minimum runtime for Metal was usually observed for smaller grid sizes with Metal over both CUDA-capable GPUs. On Metal, Pollard's Rho reached higher occupancy than ECM, while on CUDA, both algorithms were able to approach 50%, which was the theoretical limit. The mean runtime *of computational kernels* increased for both variants as the number of instances grew, and CUDA seemed to offer better performance when adjusting for grid size.

7.4 CPU and GPU implementation comparison

When observing the results for implemented variants as shown in figure 7.5, it is clear that the GPU versions did not fare well against the CPU-based implementations, with only the CUDA ECM variant being comparable but still coming off as slower. It is clear that the OpenMP-based implementations performed the best and observed a speedup when compared with sequential implementations. The GPU variants were held back by the sub-optimal arithmetic of the MAP library, together with the observed memory-related bottlenecks. When comparing GMP and MAP libraries, it was observed that for certain use cases, the MAP library could provide speedup. This, however, did not materialize for the factorization algorithms. The factorization kernels on the GPU were likely too complex, utilizing too many multi-precision variables, requiring significant resources, and causing costly spillovers and inefficient memory access patterns, which overshadowed any benefits gained from parallelization. It was observed that chained simpler operations in kernels not suffering from the mentioned bottlenecks could improve performance. Despite this, some parallelization speedup with greater grid sizes could be observed even on the GPU.

It could be observed that the ECM on GPU could provide results faster for certain input parameter choices when compared to CPU variants. This has been observed usually with low K boundaries and a larger number of attempts. With this parametrization, the solutions generated a large number of curves, tried minimal effort point multiplication with relatively small k, and moved on to generate new curves and repeat the computation. However, this parametrization was sub-optimal and did not result in the shortest observed run times. So, while the implemented ECM on GPU could, in some cases, provide better results, choosing a better parametrization was significantly preferred.

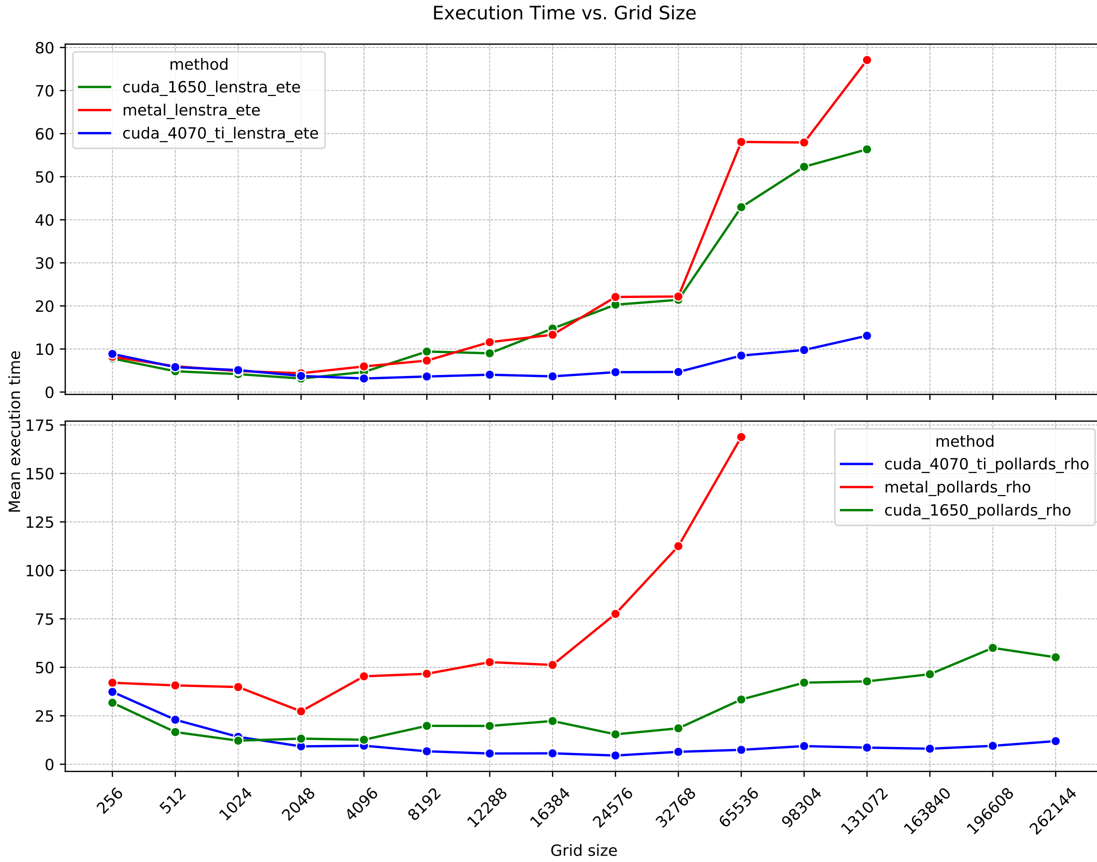

Figure 7.3 Mean execution time for CUDA and Metal compared to grid size (fixed-size variants). Shows particular configuration on 80-bit (ECM) numbers and 73-bit (Pollard's Rho) numbers.

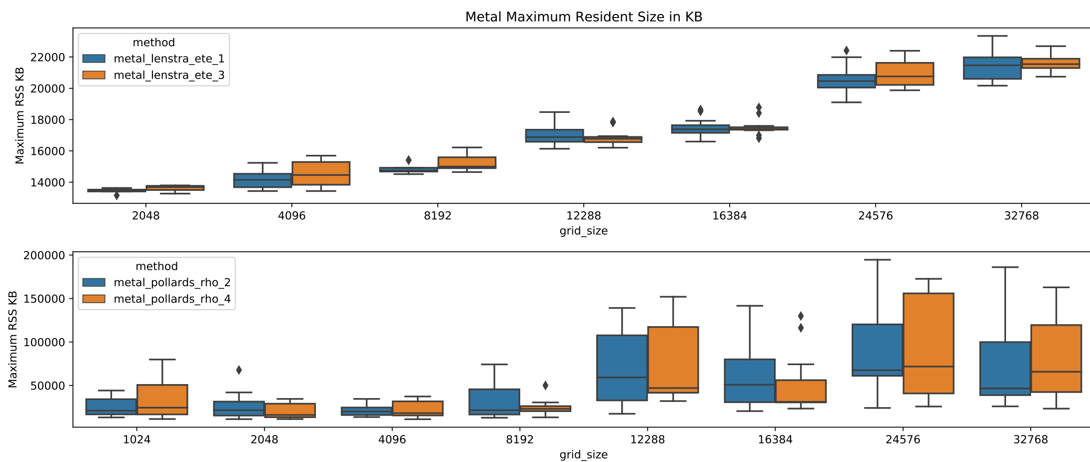

Figure 7.4 Growing maximum RSS for Metal variants on 80-bit composite input.

Method	Mean Maximum Resident Set size in KB
CUDA Lenstra ETE 2	223301.02
CUDA Pollards Rho 3	222560.25
Sage PARI	191599.95
Sage GMP-ECM	191396.52
SymPy ECM	50712.57
SymPy Pollards Rho	50069.50
Metal Pollards Rho 4	49691.79
Metal Pollards Rho 2	49327.99
Metal Lenstra ETE 1	17409.77
Metal Lenstra ETE 3	17211.88
OpenMP Lenstra ETE	851.96
OpenMP Pollards Rho	851.96
Seq ECM-ETE	851.96
Seq Pollards Rho	851.96

Table 7.1 Measured mean maximum RSS for 80-bit composite input.

An additional metric that can be observed for the implemented solutions is the memory utilization of each solution. Table 7.1 shows the mean observed maximum resident set size for 80-bit composite input across a variety of solutions. Given the scale of computation happening in the GPU variants, it is unsurprising that the GPU variants consumed much more memory. For Metal variants, growth in memory usage could be observed with increased grid sizes. This effect can be observed in 7.4 figure. For CUDA and the considered grid sizes, this was not observed, and even larger grid sizes did not result in larger maximum resident set sizes. By observing the table, it can be seen that the CUDA variant required the most memory, followed by Sage and SymPy, with Metal using, on average, less memory but being highly dependent on grid size. The implemented CPU variants required the least amount of memory. For the GPU variants, there was no large difference in memory usage for fixed and dynamic-sized variants. It should be noted that using Sage likely introduced overheads into this consideration.

7.5 SymPy, PARI and GMP-ECM implementation

The accessible SymPy library was utilized to provide a very simple baseline for the problem. Pollard's Rho and Lenstra's factorization algorithms implemented in SymPy have been wrapped in code, allowing multiple concurrent processes. The implementation is simple, given that the library abstracts away the main complexity. The code launches a desired number of processes where each has a random starting point similar to the CPU implementation. The parent process monitors whenever any factorization process has found a solution. If a result is found, the processes are terminated, and the value is returned.

Additional existing implementations to consider are accessible through Sage. The motivation to utilize Sage for additional comparison may not be immediately apparent, but doing so enables comparison with complex and advanced implementations of PARI and GMP-ECM. That is because Sage wraps around the PARI and GMP-ECM implementations mentioned in the theoretical section. Utilizing Sage provides some overhead over using the solutions directly but significantly simplifies the process. Sage essentially functions as a high-level wrapper for both of these solutions. While PARI contains Pollard's Rho and ECM implementations, it doesn't easily allow the specification and limit the factorization to one such algorithm instead of using more complex machinery to provide a solution. As the documentation claims, such a sophisticated factoring mechanism is much faster than a plain ECM implementation. Regrettably, this leads to

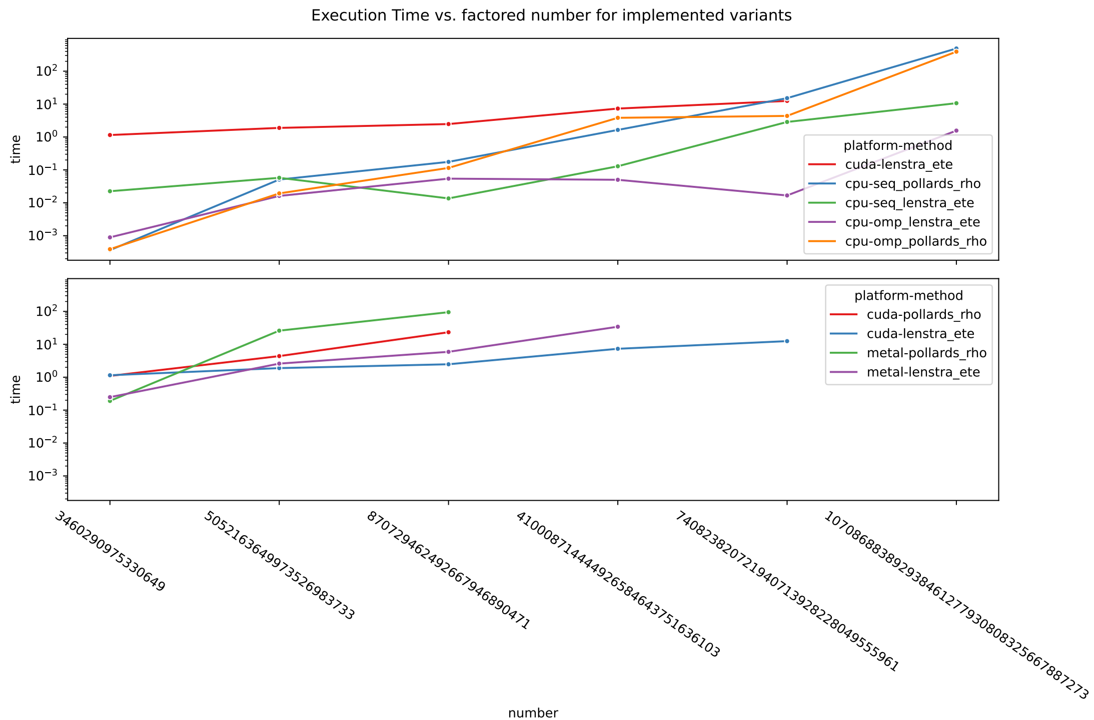

Figure 7.5 Shows log-scaled runtime comparison across implemented variants. The first plot shows CPU variants together with CUDA ECM. The second plot shows all GPU variants together.

a relative comparison of overall solutions rather than implementations of the specific algorithms.

First, if we focus on runtime as observed in figures 7.6 and 7.8, it can be seen that the PARI implementation delivers the most favorable results. This is followed by the OpenMP ECM and GMP-ECM implementations. Overall, the GMP-ECM provides better results, but for example, the 110-bit number of OpenMP *V4* provided slightly better results. The OpenMP Pollard's Rho implementation was competitive for small composites but grew quickly with increasing composite number bit-size. This is followed by CUDA ECM (which provided the best performance from the implemented GPU solutions) and SymPy ECM variants, which produced similar results. In summary, the only competitive solution for the composite numbers created in this thesis is the GMP and OpenMP-based ECM. The GPU variants failed to provide satisfactory results.

In the next figure 7.7, the methods can be observed with normalized time and growing composite size. A number of the previously observed plots made it difficult to estimate how the methods scale with growing composite size. It can be observed that the fastest-growing method is Pollard's Rho on Metal, followed by its CUDA variant. The next steepest growth is observed by the SymPy version of Pollard's Rho, which is followed by the ECM Metal variant. As for the sequential variants, Pollard's Rho provides reasonable results up to the 30-digit input. As for ECM, it has been measured up to the 40-digit input, where it significantly lagged behind the parallel variants except for OpenMP Pollard's Rho. The 33-digit composite, which corresponds to the 110-bit number, contains the last measured value for the ECM variant on CUDA, after which it grows significantly. The figure shows a rapid growth after the 33-digit composite for OpenMP Pollard's Rho, which performed the best out of the considered Pollard's Rho variants but takes significant time for larger inputs. Moving to the largest observed number, it can be seen that SymPy ECM grows significantly. For this input size, the best results are from PARI, followed by GMP-ECM, and finally, the OpenMP ECM variant. Since the figures and given commentary might be difficult to follow, the mean runtime for the discussed variants can be

Composite decimal digits Variant	16	22	24	30	33	39
Metal Pollard's Rho	0.1904	25.9796	94.8169	NaN	NaN	NaN
CUDA Pollard's Rho	1.0949	4.3833	23.2930	NaN	NaN	NaN
Metal ECM	0.2479	2.5923	5.9095	34.2284	NaN	NaN
SymPy Pollard's Rho	2.6743	1.2112	2.8055	52.5608	109.1900	NaN
CUDA ECM	1.1487	1.8910	2.4753	7.3132	12.4916	NaN
seq. Pollard's Rho	0.0004	0.0509	0.1755	1.6418	15.1476	490.1100
SymPy ECM	0.6336	0.7694	0.8961	18.9876	8.9609	66.4750
OpenMP Pollard's Rho	0.0004	0.0193	0.1149	3.8291	4.3721	391.9305
seq. ECM	0.0225	0.0576	0.0136	0.1291	2.8655	10.6502
OpenMP ECM	0.0009	0.0162	0.0543	0.0501	0.0167	1.5646
GMP-ECM	0.0081	0.0115	0.0106	0.0518	0.0409	0.2098
PARI	0.0025	0.0073	0.0055	0.0067	0.0089	0.0138

Table 7.2 Mean runtime across all solutions.

observed in 7.2.

Initially, there was an intention to compare the implementations to *ecmongpu* from [9], which would provide compassion with existing CUDA implementation. Unfortunately, when compiling the project for the system used in this thesis, the result was not fully functional, and a large portion of the included tests were failing. Multi-precision arithmetic did not work properly. The efforts to build the project correctly were unsuccessful. Since the project lacks documentation about required library versions, CUDA computing, or any other requirements and limitations, this effort was abandoned. In addition, the GMP-ECM solution offers the possibility of running the first stage on the GPU; this required building from source, which eventually resulted in compatibility issues with newer CUDA versions and the required CGBN library, which does not have explicit support for the GPU architecture of the considered system. Shortly, the attempts to enable GPU computations in GMP-ECM have been futile.

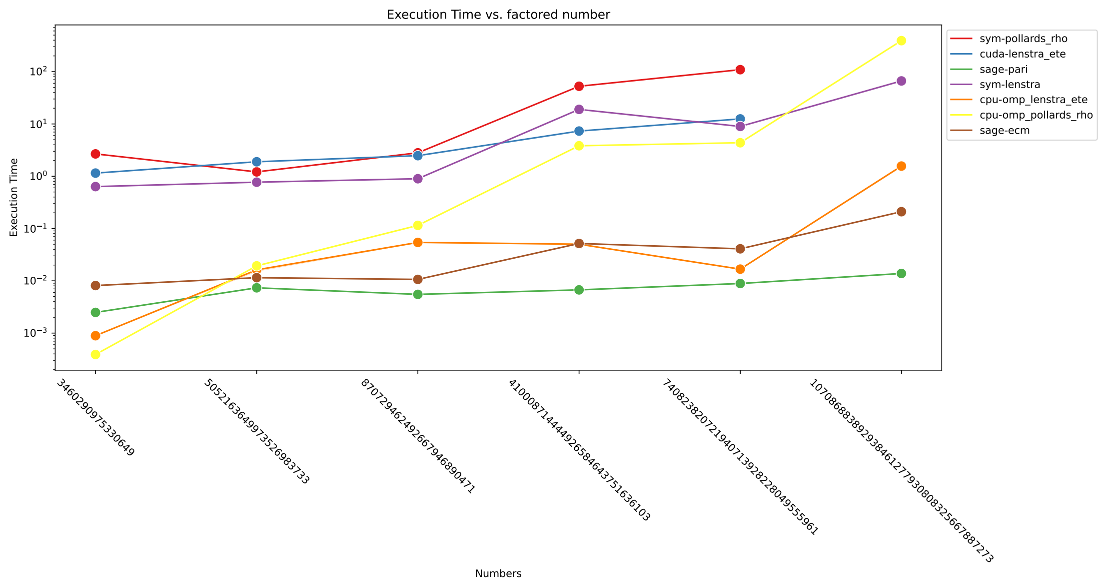

Figure 7.6 Runtime comparison of selected, better-performing implementations

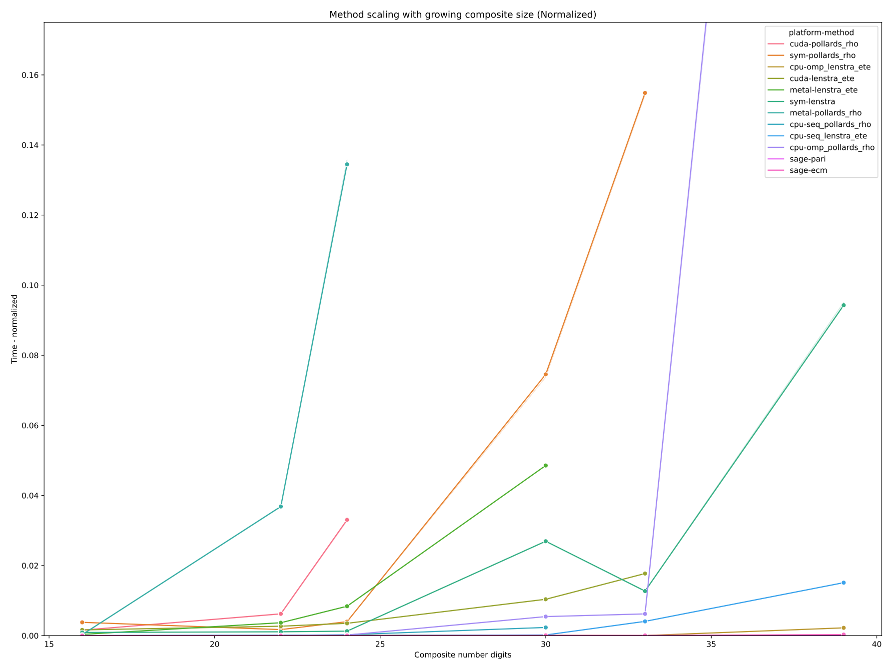

Figure 7.7 Normalized mean runtime for growing composite size.

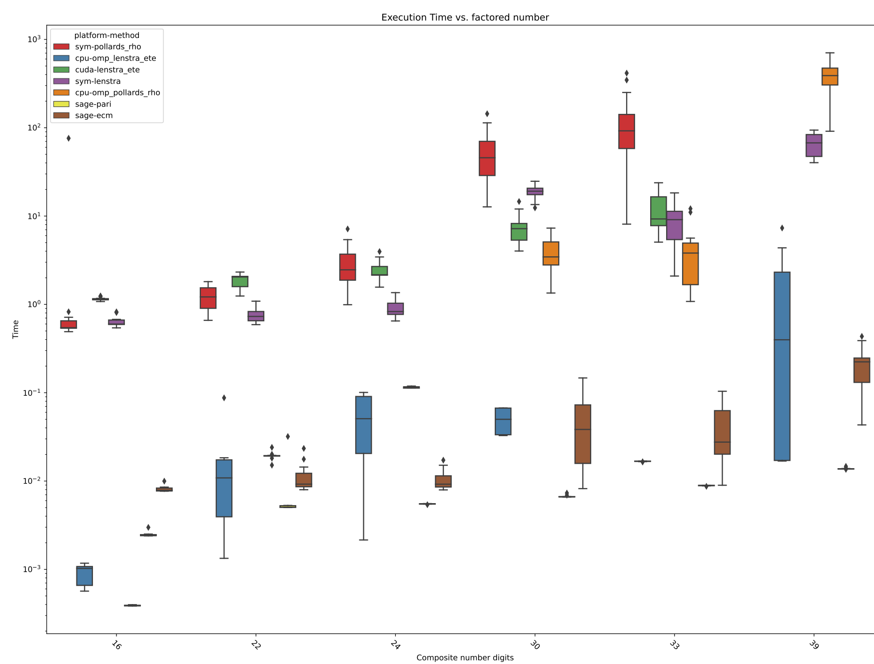

Figure 7.8 Log scaled mean runtime for growing composite size

Chapter 8
Conclusion

In this final short chapter, the overall results and measurements, as seen in this thesis, are summarized. This chapter provides a reflection on the effort in this thesis and a brief discussion of the numerous shortcomings, potential improvements, and considerations for the observed outcomes.

8.1 Outcomes

While the factorization algorithm implementations created in this thesis lag behind state-of-the-art implementations, the feasibility of implementation on the Metal API, according to the best knowledge at the time of writing of this thesis, has not been previously explored. Consequently, in order to support the Metal implementation, a new arbitrary precision library has been created for Metal, enabling various possibilities on Metal as no publicly available library would offer such functionality. This library provides a comprehensive set of arithmetic operations and can be utilized for purposes far beyond integer factorization, for which it is perhaps better suited. It has been adapted to CUDA and offers a very similar interface for both APIs, potentially simplifying cross-platform development requiring multi-precision arithmetic. The MAP library is likely the most valuable product of this thesis.

While the OpenMP-based ECM implementation seemed competitive, it offers nothing special over the more mature existing implementations. The GPU-based implementations were too constrained with memory-related bottlenecks, which stemmed from conceptually flawed design. While the previous chapter notes that the MAP library could provide speedup for use cases with a smaller set of distinct variables, the considered factorization algorithms were not successfully adapted to benefit from this.

In hindsight, the broad scope of this thesis was not beneficial. Realizing multiple implementations of two algorithms over multiple APIs requires significant time. The time consumption is also relevant in prolonging measurements, maintaining the codebase, implementing optimizations, and keeping the algorithms consistent. Introducing an algorithmic change to improve performance leads to either changing all existing CPU, Metal, and CUDA adaptations or leaving the different versions inconsistent, neither of which is ideal. Overall, too much time was invested into managing the complexity and variety of various solutions that could be used to further optimize or redesign considered approaches.

In the previous chapter, the difference in implementation and their performance has been observed, as for algorithmic comparison, it can be noted that Pollard's Rho algorithm, which has less overhead, performed well for small numbers, but with the size increasing, ECM quickly overtook and provided better results. The runtime for Pollard's Rho started to rapidly grow around 73-80 bit for GPU variants, and 99-110 bit for CPU variants. For both GPU versions,

it could be observed that Pollard's Rho could be run in greater number of concurrent instances. This is not a surprising result, given the associated overheads.

8.2 Shortcomings

In general, the approach chosen for the GPUs has been largely disappointing. While feasible and with potential speedup for specific parameter inputs, the observed issues did not make it worthwhile over simpler CPU-based solutions that utilize the existing GMP library. The most apparent shortcomings of the GPU versions are the memory-related bottlenecks. The solutions were constrained by reading and writing to slow memory as the implementation struggled to fit into and utilize faster on-chip memory. This was very taxing, given the very intensive reads and writes performed by the MAP library during arithmetic operations. This issue is magnified by the complexity of the GPU kernels, which is especially seen in the ECM implementations. For less-complex kernels, this would be a significantly smaller issue, and as was observed during various optimizations, even a small reduction in kernel complexity leads to large benefits in performance.

Beyond this bottleneck, there remains a large amount of work in optimizing the MAP library. The library can not match existing libraries, such as GMP, for a majority of use cases in complexity or speed. It lacks support for more complicated or performant algorithms for arithmetic operations and is confined to simpler algorithms. The library could be largely improved not just by introducing additional, potentially faster algorithms for operations such as multiplication, modular reductions, GCD, or faster division but also by focusing on the code aspect of the library. The later introduced fixed-size integers seemed to be a promising path for further development. The library was extended to support fixed-size integers. Still, those additions did not differ much from dynamic-sized code, definitely not using the full potential as the code could be further transformed and optimized. One such direction could be to provide operations for specific, fixed-sized inputs, such as the addition of 256, 512 or 1,024-bit integers. This could potentially allow for greater optimization, such as explicit loop unrolling or moving limb data to GPU registers, redefining the utilized structures and static indexing. All this is extremely time-consuming but would likely yield benefits. As was consistently observed across implementations (even those utilizing GMP), the arithmetic operations such as GCD or modular reduction were highly present in total computational time, making them good candidates for further improvements. The MAP library currently fills a very specific niche. Extending it to include parallelized cooperative multi-precision arithmetic (bringing it closer to the existing *CGBN* library) could be a different path for the library and would allow for greater usability.

One additional optimization could be made by further changing the memory pattern of the variables in GPU implementations. For ECM, the pattern for storing EC points was changed so that the coordinates of the points were stored closer together. This was not the case for other variables. For example, the variables X, Y, and A are accessed in Pollard's Rho in close proximity but are being stored in different Holder classes and, hence, in different sections of allocated memory, which may be stored far apart. There could be a benefit in a more strided storage approach, as could be seen in [9].

The wide scope of the thesis made it difficult to focus on different algorithmic aspects, which should, without a doubt, improve performance. Instead, the thesis focused on the computational aspect of things. One such example is relying on the simple right-to-left point multiplication method, while other, potentially faster methods were not explored. An additional focus could be on the algorithms themselves. In ECM, this only materialized by introducing the ETE variant. One such case could be a less naive generation of elliptic curves, which remained very unsophisticated in the implementations.

Appendix A

Selected algorithms for multi-precision arithmetic

This chapter showcases selected algorithms as utilized and implemented in the MAP library. The purpose of the chapter is to provide some insight into the algorithms used for multi-precision or arbitrary precision integer arithmetic.

A.1 Considerations for the selected algorithms

The shown algorithms are simplified and lack steps such as input validation or memory allocations, or other, implementation-dependant specificities or syntax. This has been done to provide a more straightforward presentation of the core aspects of the algorithms. In previous chapters the `map_int` structure was frequently mentioned. For the descriptive purposes of this chapter, an additional, simpler `mp_int` struct will be defined as shown below and referred to as an `ap_int`:

```
typedef struct {
    int used;
    int alloc;
    int sign;
    uint * dp;
} ap_int;
```

This struct will be used in the pseudo-code describing the core algorithms for multi-precision computations. Note that this chapter contains a small selection of algorithms, to get a more complete and detailed overview, please refer to the *Multi-Precision Math* [11], *Handbook of Applied Cryptography* [13] or *LibTomMath* [14] documentation which served as sources for the implemented and discussed algorithms.

A.2 Addition and subtraction

First algorithms to be discussed are the addition and subtraction, which are conceptually similar. The computational burden is on low-level addition and subtraction functions that require positive integer inputs. These functions are where the actual computation occurs, and it is up to the high-level addition and subtraction functions to correctly utilize and decide which low-level function to use, depending on the size and signs of the inputs. The pseudo-code of the low-level addition is as described in *Multi-Precision Math* [11] and can be seen in the 3, 4 algorithms, respectively.

Algorithm 3 Low-level addition

Require: Two ap ints a and b
Ensure: The unsigned addition $c = |a| + |b|$

```
1:  old.used ← c.used
2:  c.used ← max(a.used, b.used) + 1
3:  u ← 0
4:  if a.used > b.used then
5:      min ← b.used
6:      max ← a.used
7:      x ← a
8:  else
9:      min ← a.used
10:     max ← b.used
11:     x ← b
12: end if
13: for n = 0 to min − 1 do
14:     c_n ← a_n + b_n + u
15:     u ← ⌊c_n/β⌋
16:     c_n ← c_n mod β
17: end for
18: if min ≠ max then
19:     for n = min to max − 1 do
20:         c_n ← x_n + u
21:         u ← ⌊c_n/β⌋
22:         c_n ← c_n mod β
23:     end for
24: end if
25: c_max ← u
26: if old.used > max then
27:     for n = max + 1 to old.used − 1 do
28:         c_n ← 0
29:     end for
30: end if
31: Clamp excess digits in c
```

Algorithm 4 Low-level subtraction

Require: Two ap ints a and b
Ensure: The unsigned subtraction $c = |a| + |b|$

```
 1: min ← b.used
 2: max ← a.used
 3: old.used ← c.used
 4: c.used ← max
 5: u ← 0
 6: for n from 0 to min − 1 do
 7:     c_n ← a_n − b_n − u
 8:     u ← c_n ≫ (γ − 1)
 9:     c_n ← c_n mod β
10: end for
11: if min < max then
12:     for n from min to max − 1 do
13:         c_n ← a_n − u
14:         u ← c_n ≫ (γ − 1)
15:         c_n ← c_n mod β
16:     end for
17: end if
18: if old.used > max then
19:     for n from max to old.used − 1 do
20:         c_n ← 0
21:     end for
22: end if
23: Clamp excess digits of c.
```

A.3 Greatest common divisor and extended greatest common divisor

The algorithms utilized for GCD and extended GCD computation in this thesis were the Binary GCD and Binary Extended GCD algorithms as shown in *Handbook of Applied Cryptography* [13]. They require positive arbitrary-sized integer inputs and returns their greatest common divisor. The outline of Binary GCD is shown in 5, while Binary Extended GCD can be seen in 6.

A.4 Multiplication and division by two

The last shown algorithms are the division and multiplication by two, those can be seen in the 7 and 8 algorithms. Those are less complicated than multiplication between two multi-precision integers, or a multiplication of an multi-precision and standard integer.

Algorithm 5 Binary GCD

Require: Positive ap integers x and y with $x \geq y$.
Ensure: $\gcd(x, y)$.

```
 1: g ← 1
 2: while x and y are even do
 3:     x ← x/2
 4:     y ← y/2
 5:     g ← 2g
 6: end while
 7: while x ≠ 0 do
 8:     while x is even do
 9:         x ← x/2
10:     end while
11:     while y is even do
12:         y ← y/2
13:     end while
14:     t ← |x − y|/2
15:     if x ≥ y then
16:         x ← t
17:     else
18:         y ← t
19:     end if
20: end while
21: return (g · y)
```

Algorithm 6 Binary Extended GCD Algorithm

Require: Positive ap integers x and y with $x \geq y$.
Ensure: a, b, gcd(x, y).

1: $g \leftarrow 1$
2: **while** x and y are even **do**
3: $x \leftarrow x/2$
4: $y \leftarrow y/2$
5: $g \leftarrow 2g$
6: **end while**
7: $u \leftarrow x, v \leftarrow y, A \leftarrow 1, B \leftarrow 0, C \leftarrow 0, D \leftarrow 1$
8: **while** u is even **do**
9: $u \leftarrow u/2$
10: **if** $A \equiv B \equiv 0 \pmod 2$ **then**
11: $A \leftarrow A/2$
12: $B \leftarrow B/2$
13: **else**
14: $A \leftarrow (A + y)/2$
15: $B \leftarrow (B - x)/2$
16: **end if**
17: **end while**
18: **while** v is even **do**
19: $v \leftarrow v/2$
20: **if** $C \equiv D \equiv 0 \pmod 2$ **then**
21: $C \leftarrow C/2$
22: $D \leftarrow D/2$
23: **else**
24: $C \leftarrow (C + y)/2$
25: $D \leftarrow (D - x)/2$
26: **end if**
27: **end while**
28: **while** $u \neq 0$ **do**
29: **if** $u \geq v$ **then**
30: $u \leftarrow u - v$
31: $A \leftarrow A - C$
32: $B \leftarrow B - D$
33: **else**
34: $v \leftarrow v - u$
35: $C \leftarrow C - A$
36: $D \leftarrow D - B$
37: **end if**
38: **end while**
39: $a \leftarrow C, b \leftarrow D$
40: **return** $(a, b, g \cdot v)$
41:

Algorithm 7 Multiplication by two

Require: ap int a
Ensure: $b = 2a$
1: old.used $\leftarrow b$.used
2: b.used $\leftarrow a$.used
3: $r \leftarrow 0$
4: **for** n from 0 to a.used $- 1$ **do**
5: $r_r \leftarrow a_n >> (\lg(\beta) - 1)$
6: $b_n \leftarrow (a_n << 1) + r \bmod \beta$
7: $r \leftarrow r_r$
8: **end for**
9: **if** $r \neq 0$ **then**
10: b.used $+ 1 \leftarrow r$
11: b.used $\leftarrow b$.used $+ 1$
12: **end if**
13: **if** b.used $< old$.used $- 1$ **then**
14: **for** n from b.used to old.used $- 1$ **do**
15: $b_n \leftarrow 0$
16: **end for**
17: **end if**
18: b.sign $\leftarrow a$.sign

Algorithm 8 Division by two

Require: ap int a
Ensure: $b = a/2$
1: old.used $\leftarrow b$.used
2: b.used $\leftarrow a$.used
3: $r \leftarrow 0$
4: **for** n from b.used $- 1$ to 0 **do**
5: $r_r \leftarrow a_n \mod 2$
6: $b_n \leftarrow (a_n \gg 1) + (r \ll (\lg(\beta) - 1)) \mod \beta$
7: $r \leftarrow r_r$
8: **end for**
9: **if** b.used $< old$.used $- 1$ **then**
10: **for** n from b.used to old.used $- 1$ **do**
11: $b_n \leftarrow 0$
12: **end for**
13: **end if**
14: b.sign $\leftarrow a$.sign
15: Clamp excess digits of b.

Bibliography

1. BRENT, Richard P. Some Parallel Algorithms for Integer Factorisation. In: AMESTOY, Patrick; BERGER, Philippe; DAYDÉ, Michel; RUIZ, Daniel; DUFF, Iain; FRAYSSÉ, Valérie; GIRAUD, Luc (eds.). *Euro-Par'99 Parallel Processing*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 1–22. ISBN 978-3-540-48311-3.
2. LENSTRA JR, Hendrik W. Factoring integers with elliptic curves. *Annals of mathematics*. 1987, pp. 649–673.
3. GÉLIN, Alexandre; KLEINJUNG, Thorsten; LENSTRA, Arjen. Parametrizations for Families of ECM-Friendly Curves. In: 2017, pp. 165–171. ISBN 978-1-4503-5064-8. Available from DOI: 10.1145/3087604.3087606.
4. HANKERSON, Darrel; MENEZES, Alfred J; VANSTONE, Scott. *Guide to elliptic curve cryptography*. Springer Science & Business Media, 2006.
5. BERNSTEIN, Daniel J.; BIRKNER, Peter; LANGE, Tanja; PETERS, Christiane. *ECM using Edwards curves* [Cryptology ePrint Archive, Paper 2008/016]. 2008. Available also from: https://eprint.iacr.org/2008/016. https://eprint.iacr.org/2008/016.
6. BERNSTEIN, Daniel J.; BIRKNER, Peter; JOYE, Marc; LANGE, Tanja; PETERS, Christiane. Twisted Edwards Curves. In: VAUDENAY, Serge (ed.). *Progress in Cryptology – AFRICACRYPT 2008*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 389–405. ISBN 978-3-540-68164-9.
7. PARKER, DANIEL. Elliptic curves and Lenstra's factorization algorithm. *University of Chicago: REU*. 2014, vol. 2014.
8. HISIL, Huseyin; WONG, Kenneth Koon-Ho; CARTER, Gary; DAWSON, Ed. Twisted Edwards curves revisited. In: *Advances in Cryptology-ASIACRYPT 2008: 14th International Conference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008. Proceedings 14*. Springer, 2008, pp. 326–343.
9. WLOKA, Jonas; RICHTER-BROCKMANN, Jan; STAHLKE, Colin; KLEINJUNG, Thorsten; PRIPLATA, Christine; GÜNEYSU, Tim. Revisiting ECM on GPUs. In: *Cryptology and Network Security: 19th International Conference, CANS 2020, Vienna, Austria, December 14–16, 2020, Proceedings 19*. Springer, 2020, pp. 299–319.
10. POLLARD, John M. A Monte Carlo method for factorization, BIT 15 (1975), 331–334. *MR*. [N.d.], vol. 52, p. 13611.
11. ST DENIS, Tom; RASMUSSEN, Mads; ROSE, Greg. *Multi-Precision Math*. Open Communications Security, QUALCOMM Australia, 2006.
12. TEAM, The GMP Development. *Integer Internals* [https://gmplib.org/manual/Integer-Internals.html]. 2021. Accessed: March 25, 2023.

13. MENEZES, A.; OORSCHOT, P. van; VANSTONE, S. *Handbook of Applied Cryptography*. CRC Press, 1996.

14. PROJECTS, LibTom. *LibTomMath: a free open source portable number theoretic multiple-precision integer (MPI) library* [https://github.com/libtom/libtommath]. 2021. Accessed: March 25, 2023.

15. TEAM, The GMP Development. *GMP Contributors* [https://gmplib.org/manual/Contributors.html]. 2021. Accessed: March 25, 2023.

16. TEAM, The GMP Development. *GMP MP Manual* [https://gmplib.org/gmp-man-6.1.0.pdf]. 2015. Accessed: Nov 20, 2023.

17. *PARI/GP version 2.13.4*. Univ. Bordeaux: The PARI Group, 2022. available from http://pari.math.u-bordeaux.fr/.

18. *ZIMMERMANN Paul / ecm · GitLab — gitlab.inria.fr* [https://gitlab.inria.fr/zimmerma/ecm]. [N.d.]. [Accessed 21-11-2023].

19. NVLABS. *CGBN: CUDA Multi-Precision Integer Arithmetic* [https://github.com/NVlabs/CGBN]. accessed March 14, 2023.

20. MEURER, Aaron; SMITH, Christopher P.; PAPROCKI, Mateusz; ČERTÍK, Ondřej; KIRPICHEV, Sergey B.; ROCKLIN, Matthew; KUMAR, AMiT; IVANOV, Sergiu; MOORE, Jason K.; SINGH, Sartaj; RATHNAYAKE, Thilina; VIG, Sean; GRANGER, Brian E.; MULLER, Richard P.; BONAZZI, Francesco; GUPTA, Harsh; VATS, Shivam; JOHANSSON, Fredrik; PEDREGOSA, Fabian; CURRY, Matthew J.; TERREL, Andy R.; ROUČKA, Štěpán; SABOO, Ashutosh; FERNANDO, Isuru; KULAL, Sumith; CIMRMAN, Robert; SCOPATZ, Anthony. SymPy: symbolic computing in Python. *PeerJ Computer Science*. 2017, vol. 3, e103. ISSN 2376-5992. Available from DOI: 10.7717/peerj-cs.103.

21. *Number Theory - SymPy 1.12 documentation — docs.sympy.org* [https://docs.sympy.org/latest/modules/ntheory.html]. [N.d.]. [Accessed 21-11-2023].

22. OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP* [https://www.openmp.org]. Accessed on March 24, 2023.

23. OPENMP ARCHITECTURE REVIEW BOARD. *OpenMP API Specification and Reference Guide* [https://www.openmp.org/wp-content/uploads/OpenMPRefCard-5-2-web.pdf]. 2021.

24. PHARR, Matt; FERNANDO, Randima. *General-Purpose Computation on GPUs: A Primer* [https://developer.nvidia.com/gpugems/gpugems2/part-iv-general-purpose-computation-gpus-primer]. 2005. [Online; accessed 25-Mar-2023].

25. NVIDIA CORPORATION. *NVIDIA TensorFlow* [https://www.nvidia.com/en-sg/data-center/gpu-accelerated-applications/tensorflow/]. 2021. Accessed on March 26, 2023.

26. APPLE INC. *Metal Feature Set Tables* [https://developer.apple.com/metal/Metal-Feature-Set-Tables.pdf]. 2021. Accessed on March 26, 2023.

27. LU, Mian; HE, Bingsheng; LUO, Qiong. Supporting Extended Precision on Graphics Processors. In: *Proceedings of the Sixth International Workshop on Data Management on New Hardware*. Indianapolis, Indiana: ACM, 2010, pp. 19–26. DaMoN '10. ISBN 978-1-4503-0189-3. Available from DOI: 10.1145/1869389.1869392.

28. LANGER, Bernhard. *Arbitrary-Precision Arithmetics on the GPU*. 2017. Bachelor's Thesis. Vienna University of Technology. Supervised by Thomas AUZINGER.

29. PETROUS, Petr. *Accelerating Modular Arithmetic on the GPU* [https://dspace.cvut.cz/bitstream/handle/10467/65185/F8-DP-2016-Petrous-Petr-thesis.pdf?sequence=1&isAllowed=y]. Prague, Czech Republic, 2016.

30. HARRIS, Mark. GPU Flow Control Idioms. In: PHARR, Matt (ed.). *GPU Gems 2* [https://developer.nvidia.com/gpugems/gpugems2/part-iv-general-purpose-computation-gpus-primer/chapter-34-gpu-flow-control-idioms]. Addison-Wesley, 2005, chap. 34, pp. 609–622. ISBN 0-321-33559-7. Accessed on March 26, 2023.
31. NVIDIA CORPORATION. *CUDA C Programming Guide*. NVIDIA Corporation, 2022. Available also from: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.
32. NVIDIA CORPORATION. *Volta Architecture Whitepaper* [https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf]. 2017. Accessed on March 26, 2023.
33. *Inside Volta: The World's Most Advanced Data Center GPU — NVIDIA Technical Blog — developer.nvidia.com* [https://developer.nvidia.com/blog/inside-volta/]. [N.d.]. [Accessed 23-11-2023].
34. INC., Apple. *Creating Threads and Threadgroups* [https://developer.apple.com/documentation/metal/compute_passes/creating_threads_and_threadgroups?language=objc]. 2021. [Online; accessed 14 March 2023].
35. CORPORATION, NVIDIA. *CUDA C Best Practices Guide* [https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html]. 2021. Accessed on March 29, 2023.
36. STEVE RENNICH, NVIDIA. *CUDA C/C++ Streams and Concurrency* [https://developer.download.nvidia.com/CUDA/training/StreamsAndConcurrencyWebinar.pdf]. [N.d.]. [Accessed 23-11-2023].
37. INC., Apple. *Metal* [https://developer.apple.com/documentation/metal?language=objc]. 2021. [Online; accessed 14 March 2023].
38. INC., Apple. *Metal Shading Language Specification* [https://developer.apple.com/metal/Metal-Shading-Language-Specification.pdf]. 2021. [Online; accessed 14 March 2023].
39. INC., Apple. *Performing Calculations on a GPU* [https://developer.apple.com/documentation/metal/performing_calculations_on_a_gpu?language=objc]. 2021. [Online; accessed 14 March 2023].
40. APPLE INC. *Metal for C++* [https://developer.apple.com/metal/cpp/]. [N.d.]. [Accessed: March 14, 2023].
41. *Optimize Metal Performance for Apple silicon Macs - WWDC20 - Videos - Apple Developer — developer.apple.com* [https://developer.apple.com/videos/play/wwdc2020/10632/?time=2420]. [N.d.]. [Accessed 23-11-2023].
42. INC., Apple. *Metal Compute on MacBook Pro* [Online video]. Apple Inc., 2019. Available also from: https://developer.apple.com/videos/play/tech-talks/10580.
43. *Scale compute workloads across Apple GPUs*. Apple Inc., 2022. Available also from: https://developer.apple.com/videos/play/wwdc2022/10159/. Accessed on March 28, 2023.
44. INC., Apple. *Setting Up a Command Structure* [https://developer.apple.com/documentation/metal/gpu_devices_and_work_submission/setting_up_a_command_structure?language=objc]. Apple Inc., accessed March 14, 2023.
45. DVORAK, Jakub. *Faktorizace pomocí eliptických křivek* [https://dspace.cvut.cz/bitstream/handle/10467/92951/F8-DP-2021-Dvorak-Jakub-thesis.pdf?sequence=-1&isAllowed=y]. 2021. Master's thesis, Czech Technical University in Prague.
46. VICTOR, Youssef. *Loki Random Number Generator*. 2017. Available also from: https://github.com/YoussefV/Loki.

47. MOHANTY, Siddhant; MOHANTY, A K; CARMINATI, F. Efficient pseudo-random number generation for monte-carlo simulations using graphic processors. *Journal of Physics: Conference Series*. 2012, vol. 368, no. 1, p. 012024. Available from DOI: 10.1088/1742-6596/368/1/012024.
48. APPLE INC. *Advanced Metal Shader Optimization*. 2016. Available also from: https://developer.apple.com/videos/play/wwdc2016/606/. Accessed on November 3, 2023.
49. MICIKEVICIUS, Paulius. *Local Memory and Register Spilling*. 2011. Tech. rep. NVIDIA. Available also from: https://developer.download.nvidia.com/CUDA/training/register_spilling.pdf.

Contents of enclosed CD

```
thesis.pdf ............................ Thesis text PDF
└── source ............................ Source code and measured data
```