

Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH



Instytut Telekomunikacji

Praca dyplomowa inżynierska

na kierunku Telekomunikacja
w specjalności Techniki Teleinformatyczne

Akceleracja kryptoanalizy ECDLP przy użyciu GPU

Andrzej Tłomak

Numer albumu 311450

promotor

dr hab. inż. Mariusz Rawski

WARSZAWA 2025

Akceleracja kryptoanalizy ECDLP przy użyciu GPU

Streszczenie. Problem logarytmu dyskretnego na krzywych eliptycznych (ECDLP) stanowi podstawę bezpieczeństwa wielu współczesnych systemów kryptograficznych. Ze względu na jego wysoką złożoność obliczeniową, próby złamania ECDLP często wymagają zastosowania wyspecjalizowanego sprzętu w celu akceleracji obliczeń. W niniejszej pracy zaprezentowano implementację systemu rozwiązującego ECDLP z wykorzystaniem GPU oraz technologii CUDA. System został zoptymalizowany do działania na krzywej ECCp79 z Certicom Challenge, osiągając wydajność na poziomie 87,7 miliona operacji na sekundę, co pozwala na rozwiązanie problemu ECDLP na krzywej ECCp79 w ciągu 3 godzin.

Słowa kluczowe: Krzywe eliptyczne, ECDLP, Kryptoanaliza, CUDA, Algorytm rho Pollard'a

Acceleration of ECDLP Cryptanalysis Using GPUs

Abstract. The discrete logarithm problem on elliptic curves (ECDLP) forms the foundation of security for many modern cryptographic systems. Due to its high computational complexity, attempts to break ECDLP often require specialized hardware to accelerate computations. This paper presents the implementation of a system for solving ECDLP using GPU acceleration and CUDA technology. The system has been optimized for the ECCp79 curve from the Certicom Challenge, achieving a performance of 87.7 million operations per second, enabling the ECDLP over ECCp79 curve to be solved within 3 hours.

Keywords: Elliptic curves, ECDLP, Cryptanalysis, CUDA, rho Pollard algorithm

Spis treści

1. Wprowadzenie	7
1.1. Cel pracy	7
2. Wstęp teoretyczny	9
2.1. Procesor graficzny	9
2.1.1. Model programowania CUDA	9
2.1.2. Hierarchia pamięci	10
2.2. Krzywe eliptyczne	10
2.2.1. Dodawanie punktów na krzywej eliptycznej	11
2.2.2. Krzywe eliptyczne na ciele skończonym	12
2.2.3. Problem logarytmu dyskretnego	13
2.2.4. Problem logarytmu dyskretnego na krzywej eliptycznej	13
2.3. Algorytm Rho Pollard'a	14
2.3.1. Równoległy algorytm Rho Pollard'a	15
2.3.2. Adding Walk	15
3. Koncepcja	17
3.1. Klient	17
3.2. Serwer	18
3.3. Przebieg działania	18
4. Implementacja	20
4.1. Oprogramowanie oraz platforma sprzętowa	20
4.1.1. Platforma sprzętowa	20
4.1.2. Program po stronie CPU	20
4.1.3. System budowania oraz kompilacja CUDA C++	20
4.1.4. Testy	20
4.2. Program klienta Python	21
4.3. Program klienta CUDA	21
4.4. Arytmetyka na ciele	22
4.4.1. Reprezentacja długich liczb	22
4.4.2. Operacje na długich liczbach	23
4.4.3. Dodawanie i odejmowanie modulo	24
4.4.4. Mnożenie modulo	24
4.4.5. Odwrotność modulo	24
4.4.6. Biblioteka CGBN	26
4.5. Funkcja iterująca	27
4.5.1. Rozpoczęcie obliczeń	27
4.5.2. Wstępnie obliczone punkty	27
4.5.3. Punkty wyróżnione	28

4.5.4. Obliczanie odwrotności w seriach	29
4.6. Tail effect	31
4.7. Program serwera	33
4.7.1. Generacja punktów startowych	34
4.7.2. Kolizje	35
4.8. Logowanie	36
5. Wyniki	38
5.1. Testy implementacji	38
5.2. Wydajność	38
5.3. Rozwiązanie ECDLP dla krzywej ECCp79	39
5.3.1. Weryfikacja wyników za pomocą SageMath.	40
5.4. Podsumowanie wyników.	41
5.5. Porównanie wyników	42
6. Podsumowanie oraz dalsze usprawnienia	44
6.1. Krytyczna ocena oraz dalsze usprawnienia	44
Bibliografia	46
Wykaz symboli i skrótów	49
Spis wydruków	49
Spis załączników	50

1. Wprowadzenie

Wraz z rosnącą potrzebą zapewnienia poufności i integralności danych w systemach informatycznych, kryptografia stała się jednym z filarów współczesnych rozwiązań bezpieczeństwa. Kryptografia oparta na krzywych eliptycznych **ECC** (*Elliptic Curve Cryptography*) jest obecnie jednym z najważniejszych standardów w tej dziedzinie, oferując wysoki poziom bezpieczeństwa przy znacznie mniejszych rozmiarach kluczy w porównaniu do tradycyjnych algorytmów, takich jak RSA [1, 2]. Z tego powodu ECC znajduje szerokie zastosowanie w systemach o ograniczonych zasobach, urządzeniach IoT (*Internet of Things*) czy urządzeniach mobilnych [32, 17].

Bezpieczeństwo ECC opiera się na trudności rozwiązania problemu logarytmu dyskretnego na krzywych eliptycznych. Jest to problem matematyczny, który w praktyce nie ma efektywnego rozwiązania w rozsądnym czasie przy użyciu współczesnych metod obliczeniowych. Do tej pory nie opracowano algorytmu zdolnego rozwiązać ten problem w czasie podwykładniczym na komputerach klasycznych, dlatego kryptoanaliza ECC wymaga dużych zasobów obliczeniowych [23]. Z tego powodu, często wykorzystywane w tej roli są układy GPU [7, 26, 4], FPGA [33, 22, 16, 21] czy nawet konsole dla graczy [6].

Gwałtowny rozwój kart graficznych **GPU** (*Graphic Processing Unit*) oraz technologii **CUDA** (*Compute Unified Device Architecture*) znacznie zwiększył możliwości przeprowadzania takich analiz. Choć rozwój ten był w dużej mierze napędzany zapotrzebowaniem związanym z uczeniem maszynowym, jego zastosowanie w innych dziedzinach, takich jak kryptoanaliza jest równie istotny.

Karty graficzne, dzięki architekturze zoptymalizowanej do równoległych obliczeń, stały się narzędziem o kluczowym znaczeniu w dziedzinach wymagających intensywnego przetwarzania danych. Platforma CUDA, opracowana przez firmę NVIDIA, umożliwia wykorzystanie mocy obliczeniowej GPU do realizacji zadań takich jak implementacja algorytmów kryptograficznych oraz analiza ich odporności na ataki. CUDA pozwala na masywnie równoległe przetwarzanie danych, co ma kluczowe znaczenie w implementacji algorytmu Rho Pollard'a, który w wersji równoległej, pozwala na efektywne wykorzystanie możliwości GPU.

Chociaż GPU nie oferują wydajności i efektywności energetycznej porównywalnej z dedykowanymi układami, takimi jak FPGA czy ASIC, ich dostępność i stosunkowo niski koszt czyni je atrakcyjnym wyborem do obliczeń kryptograficznych. W obliczu rosnącej mocy obliczeniowej GPU oraz ich szerokiej dostępności, ciągła analiza odporności algorytmów kryptograficznych na ataki staje się jeszcze ważniejszym elementem ich rozwoju.

1.1. Cel pracy

Celem tej pracy jest realizacja systemu korzystającego z koprocessora GPU, w celu przyspieszenia obliczeń przy rozwiązywaniu problemu logarytmu dyskretnego. Implemen-

tacja została zoptymalizowana i dostosowana do obliczenia logarytmu dyskretnego na krzywej eliptycznej ECCp-79, zaproponowanej w wyzwaniu *The Certicom ECC Challenge* [8].

Głównymi elementami stworzonego systemu jest program klienta wykonującego część algorytmu Rho Pollard'a na karcie graficznej Nvidia z wykorzystaniem technologii CUDA i języka programowania CUDA C++, oraz program serwera działający na CPU, odpowiedzialny za zarządzanie klientami i zbieranie wyników. Wyniki implementacji zostały porównane z innymi pracami których celem była kryptoanaliza ECC z wykorzystaniem akceleracji sprzętowej.

2. Wstęp teoretyczny

2.1. Procesor graficzny

Procesory graficzne są specjalnym rodzajem procesorów, pierwotnie stworzonych do akceleracji obliczeń graficznych. Obliczenia te charakteryzują się dużą liczbą względnie prostych, podobnych operacji, które mogą być przeprowadzone równolegle. Taki model obliczeń nosi nazwę **SIMD** (*Single Instruction Multiple Data*) i oznacza równoległe wykonywanie tej samej operacji dla wielu różnych danych wejściowych. Współczesne karty graficzne są zaprojektowane do wykorzystania ich możliwości równoległych obliczeń w znacznie szerszym obszarze niż pierwotny cel akceleracji grafiki komputerowej. Technologie *OpenCL* oraz *CUDA* umożliwiają programowanie GPU w dziedzinach takich jak obliczenia naukowe, machine learning czy kryptografia.

W pracy, w celu przeprowadzenia ataku na krzywą eliptyczną, wykorzystano algorytm *Rho Pollard'a*, który z niewielkimi modyfikacjami można bardzo skutecznie wykonywać równolegle. Z tego powodu wykorzystanie GPU do kryptoanalizy, pozwala znacznie skrócić czas obliczeń. W celu stworzenia programu na GPU użyto technologii Nvidia *CUDA*, głównie ze względu na znacznie lepiej rozwinięty ekosystem oraz dostępność materiałów w internecie. Standard *OpenCL* w przeciwieństwie do *CUDA*, jest tworzony na zasadach *open-source* oraz może zostać wykorzystany do programowania kart graficznych innych producentów. Niestety jest zauważalnie gorzej wspierany w przypadku kart graficznych Nvidia.

2.1.1. Model programowania CUDA

Program napisany w *CUDA*, składa się z jednego lub więcej *kernel'i* - funkcji programu, która będzie się wykonywać równolegle na GPU, na każdym z uruchomionych wątków. Wątki są grupowane w *bloki*, które mogą się składać z 1 do 1024 wątków w przypadku *CUDA 7.5* [13]. Następnie bloki są uruchamiane na dostępnych jednostkach **SM** (*streaming multiprocessor*). Karty graficzne Nvidia składają się zazwyczaj z kilkunastu SM, które mogą równocześnie uruchomić wiele bloków, co pozwala na równoczesne wykonywanie kilkuset wątków. Wątki w ramach jednego bloku dzielą pamięć współdzieloną *shared memory* oraz wykonują się równocześnie. Możliwe jest uruchomienie znacznie większej ilości bloków, niż może się jednocześnie wykonywać na dostępnych SM. W takiej sytuacji niektóre bloki będą oczekiwać na wolne zasoby aż poprzednie nie zakończą pracy. Dodatkowo, wątki w ramach bloku wykonują tę samą instrukcję w grupach po 32, nazywanych *warp'ami*. W przypadku architektury SIMD ważne jest unikanie długich segmentów warunkowych, ponieważ skutkuje to sekwencyjnymi obliczeniami. W sytuacji gdy następuje rozgałęzienie kodu - *branching*, część wątków w *warp'ie* musi czekać na pozostałe, skutkując sekwencyjnym wykonywaniem kodu. Jest to szczególnie istotne dla wydajności działania programu na GPU [13].

2.1.2. Hierarchia pamięci

Kolejnym ważnym elementem który mocno wpływa na wydajność programu, jest odpowiedni dostęp do pamięci. Tak samo jak w zwykłym procesorze, GPU ma kilka warstw pamięci, które różnią się rozmiarem oraz czasem dostępu. W CUDA można wyróżnić 3 najważniejsze warstwy pamięci:

- Rejestry - najszybszy rodzaj pamięci, dostępny w ramach pojedynczego wątku. Ilość rejestrów dla każdego wątku jest jednak mocno ograniczona w przypadku uruchomienia wielu wątków równocześnie. Jeżeli program używa więcej rejestrów niż jest dostępne, może wystąpić *register spilling*, który wprowadza znaczne opóźnienia.
- *Shared memory* - szybka pamięć współdzielona, która jest wspólna dla wątków w danym bloku. Jest ona ograniczona przez ilość pamięci w SM. Na karcie graficznej z CUDA 7.5 jej rozmiar wynosi 64 KB [13]. Stosowana jest w przypadku, gdy wiele wątków musi się ze sobą komunikować lub w celu cache'owania danych i ograniczenia dostępu do znacznie wolniejszej pamięci globalnej. Zbyt duży rozmiar wykorzystywanej pamięci współdzielonej ogranicza ilość bloków które mogą jednocześnie się wykonywać na jednym SM.
- *Global memory* - pamięć globalna DRAM, najwolniejsza oraz największa ze wszystkich warstw. Wykorzystywana głównie w celu komunikacji karty graficznej z procesorem, w celu przesyłania danych do obliczeń oraz zapisu wyników.

Dostępne są również dodatkowe rodzaje takie jak *texture memory* oraz *constant memory*, które różnią się optymalnym sposobem dostępu, jednak nie są wykorzystywane w tej pracy.

2.2. Krzywe eliptyczne

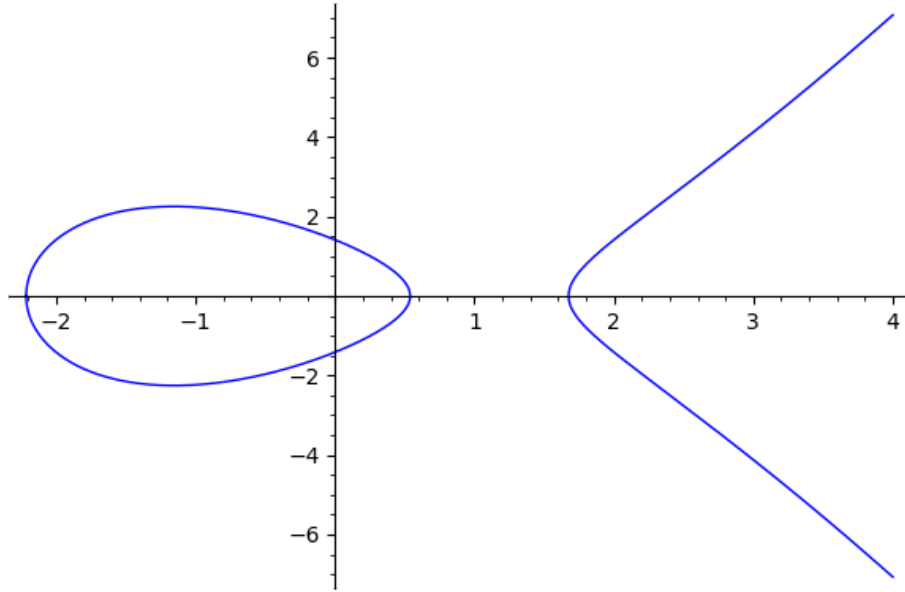
Zakładając, że ciało \mathbb{K} ma charakterystykę różną od 2 i 3, oraz że stałe $a, b \in \mathbb{K}$ spełniają warunek:

$$4a^3 + 27b^2 \neq 0$$

nieosobliwą krzywą eliptyczną nad ciałem \mathbb{K} definiuje się jako zbiór punktów $(x, y) \in \mathbb{K} \times \mathbb{K}$, spełniających równanie:

$$y^2 = x^3 + ax + b$$

wraz ze specjalnym punktem w nieskończoności \mathcal{O} , który pełni rolę elementu neutralnego w działaniach grupowych [30].



Rys. 2.1. Krzywa eliptyczna $y^2 = x^3 - 4x + 2$ nad ciałem liczb rzeczywistych

Krzywe eliptyczne zdefiniowane na liczbach rzeczywistych (rys. 2.1) nie są kluczowe w systemach kryptograficznych [30], ale takie ustawienia pozwalają na prostsze przedstawienie niektórych zagadnień np. dodawanie punktów na krzywej.

2.2.1. Dodawanie punktów na krzywej eliptycznej

Odpowiednie zdefiniowanie operacji dodawania punktów na krzywej eliptycznej pozwala otrzymać grupę abelową, złożoną z punktów krzywej oraz punktu w nieskończoności jako elementu neutralnego.

Geometrycznie, dodawanie punktów na krzywej eliptycznej nad ciałem liczb rzeczywistych można przedstawić jako połączenie dwóch punktów P i Q prostą linią, która przecina krzywą w trzecim punkcie, R' . Następnie, wynikowy punkt R , będący sumą $P + Q$, znajdujemy przez odbicie punktu R' względem osi x (rys. 2.2). W przypadku podwojenia punktu, czyli dodawania punktu P do siebie samego, rysujemy styczną do krzywej w punkcie P , która przecina krzywą w nowym punkcie. Odbicie tego punktu względem osi x daje nam wynik $2P$ [12][30].

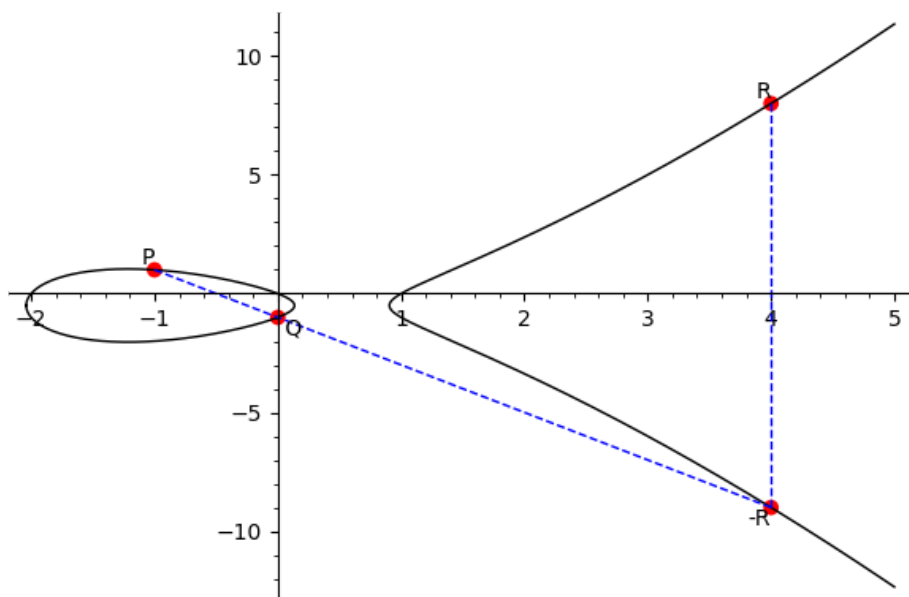
Definiując dodawanie punktów na krzywej eliptycznej w sposób algebraiczny, otrzymujemy następujące wzory:

1. Przypadek, gdy $P \neq Q$:

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}, \quad (1)$$

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

$$y_3 = \lambda(x_1 - x_3) - y_1 \quad (3)$$



Rys. 2.2. $P + Q$ na krzywej eliptycznej $y^2 + y = x^3 - x^2 + 2x$

2. Przypadek, gdy $P = Q$:

$$\lambda = \frac{3x_1^2 + a}{2y_1},$$

$$x_3 = \lambda^2 - 2x_1,$$

$$y_3 = \lambda(x_1 - x_3) - y_1$$

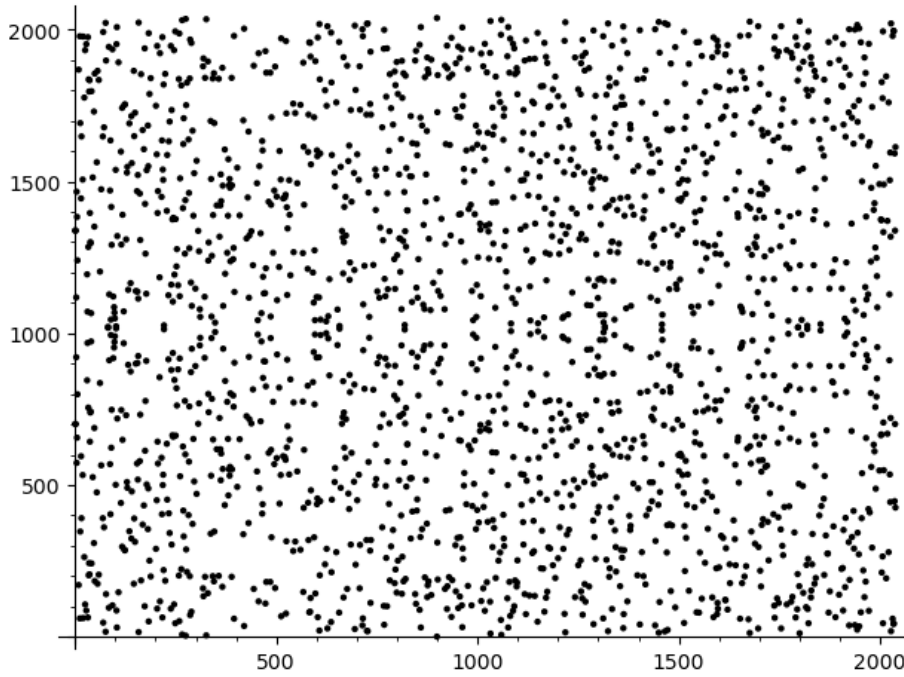
3. Szczególny przypadek, gdy $P = -Q$:

$$P + (-P) = \mathcal{O}$$

Dodatkowo odwrotność punktu na krzywej P definiujemy jako $-P = (x, -y)$ [30].

2.2.2. Krzywe eliptyczne na ciele skończonym

Krzywe eliptyczne zdefiniowane na ciele skończonym F_p oraz F_{p^n} mają kluczowe znaczenie w kryptografii [30]. Praca ta skupia się wyłącznie na krzywych zdefiniowanych na ciele skończonym F_p gdzie p jest liczbą pierwszą. Wykres krzywej eliptycznej nad ciałem F_p (rys. 2.3), nie przypomina krzywej zdefiniowanej na liczbach rzeczywistych. Krzywa taka składa się z dyskretnych punktów, których współrzędne należą do ciała na którym jest opisana. Operacje na krzywej nad ciałem skończonym są zdefiniowane za pomocą tych samych wzorów algebraicznych, co w przypadku ciała liczb rzeczywistych, jednak wszystkie działania są wykonywane na ciele F_p .



Rys. 2.3. Krzywa eliptyczna $y^2 = x^3 - 4 + 2$ nad $GF(2^{11} - 9)$

2.2.3. Problem logarytmu dyskretnego

Problem logarytmu dyskretnego **DLP** (*Discrete Logarithm Problem*) jest podstawą kryptosystemów opartych na grupach. Jednymi z bardziej znanych są kryptosystem El-Gamala oraz protokół wymiany kluczy Diffie-Hellmana'a [30, 5]. Problem logarytmu dyskretnego można zdefiniować na grupach cyklicznych, zarówno na grupie multiplikatywnej (\mathbb{G}, \cdot) jak i na grupie addytywnej $(\mathbb{G}, +)$, przy odpowiednim zdefiniowaniu działań grupowych.

Jeżeli G jest multiplikatywną grupą cykliczną a γ jej generatorem, to logarytmem dyskretnym elementu $\alpha \in G$ nazywamy najmniejszą nieujemną liczbę całkowitą x taką, że:

$$x = \log_{\gamma} \alpha$$

Uważa się, że problem logarytmu dyskretnego jest trudny obliczeniowo, ponieważ nie istnieje algorytm, który znajduje x w czasie wielomianowym[30].

2.2.4. Problem logarytmu dyskretnego na krzywej eliptycznej

W przypadku kryptografii opartej na krzywych eliptycznych, DLP dotyczy cyklicznej grupy addytywnej $(\mathbb{E}, +)$ zdefiniowanej na krzywej eliptycznej. Aby utworzyć taką grupę, wybieramy punkt P na krzywej eliptycznej \mathbb{E} , który będzie generatorem grupy. Wtedy grupa addytywna \mathbb{E} jest generowana przez kolejne potęgi punktu P :

$$\langle P \rangle = \{P, 2P, 3P, \dots, nP = \mathcal{O}\}$$

W takim przypadku, ponieważ operacją na grupie jest dodawanie modulo n , to działanie potęgowania przedstawia się jako zwielokrotnienie punktu P :

$$x \cdot P = Q \pmod{n}$$

Analogicznie do problemu logarytmu dyskretnego na grupach multiplikatywnych, problem logarytmu dyskretnego na krzywej eliptycznej polega na znalezieniu x .

Przy odpowiednim wyborze grupy addytywnej, rozwiązanie problemu logarytmu dyskretnego, tj. znalezienie x , jest trudne [30].

2.3. Algorytm Rho Pollard'a

Zaproponowany przez Johna Pollard'a w 1978 roku [27], algorytm Rho Pollard'a opiera się na wykorzystaniu paradoksu dnia urodzin w celu znalezienia logarytmu dyskretnego. Pozwala on na znalezienie rozwiązania w czasie $O(\sqrt{n})$, jednak jest to jedynie czas *oczekiwany*, ze względu na losową naturę algorytmu [5]. W porównaniu do innego znanego algorytmu, Baby-Step Giant-Step [30], algorytm Rho Pollard'a jest bardziej efektywny pamięciowo, nie wymagając przestrzeni $O(\sqrt{n})$ a jedynie $O(1)$ w wersji sekwencyjnej [30][5].

Idea algorytmu polega na losowym błędzeniu po krzywej eliptycznej w celu znalezienia kolizji dwóch punktów, które spełniają równanie:

$$aP + bQ = a'P + b'Q \pmod{n}$$

gdzie P jest generatorem grupy cyklicznej rzędu n na krzywej eliptycznej \mathbb{E} oraz $x \cdot P = Q$, więc x jest szukanym rozwiązaniem problemu. Gdy znajdziemy kolizję, odpowiednio przekształcając powyższe równanie, możemy znaleźć x :

$$x \equiv \frac{(a - a')}{(b' - b)} \pmod{n}$$

Klasyczny algorytm Rho Pollard'a, oparty o poszukiwanie cyklu, aby znaleźć kolizję, iteracyjnie oblicza parę trójek: (R_i, a_i, b_i) oraz (R_j, a_j, b_j) gdzie $R_j, R_i \in \mathbb{E}$, które spełniają własność $R = aP + bQ$:

$$f(R, a, b) = \begin{cases} (R + P, a, b + 1) & \text{if } R \in S_1, \\ (2R, 2a, 2b) & \text{if } R \in S_2, \\ (R + Q, a + 1, b) & \text{if } R \in S_3, \end{cases}$$

natomiast $S_1 \cup S_2 \cup S_3$ jest podziałem \mathbb{E} na trzy podzbiory, które powinny być podobnej wielkości. Ponieważ R jest punktem na krzywej eliptycznej a nie liczbą całkowitą, często stosowanym sposobem podziału różnych wartości R na trzy zbiory, jest obliczanie operacji modulo 3 ze współrzędnej x . Aby z kolejno obliczanych trójek znaleźć kolizję punktów,

zazwyczaj stosuje się algorytm poszukiwania cyklu Floyd’a. W takim przypadku, w każdej iteracji oblicza się trójki: (R_i, a_i, b_i) oraz (R_{2i}, a_{2i}, b_{2i}) , aż do znalezienia kolizji $R_i = R_{2i}$.

Sekwencyjna wersja algorytmu słabo się skaluje w przypadku zwiększania ilości równoległe działających procesorów, osiągając jedynie przyśpieszenie rzędu $O(\sqrt{m})$ dla m procesorów [25]. Dlatego w swojej pracy wykorzystałem równoległą wersję algorytmu, zaproponowaną przez Van Oorschota i Wienera [25].

2.3.1. Równoległy algorytm Rho Pollard’a

Równoległa wersja algorytmu Rho Pollard’a, zakłada zastosowanie wielu równoległe działających procesorów, które niezależnie od siebie wykonują *spacer losowy* po krzywej eliptycznej, w poszukiwaniu *punktów wyróżnionych*. Gdy znajdą taki punkt, przekazują go do serwera centralnego, który odpowiada za gromadzenie znalezionych punktów wyróżnionych oraz poszukiwanie kolizji między nimi.

Cecha określająca czy dany punkt na krzywej jest wyróżniony, powinna być łatwo weryfikowalna i tania obliczeniowo, ponieważ sprawdzenie czy dany punkt jest wyróżniony, występuje w każdej iteracji algorytmu.

Często wybieranym sposobem sprawdzenia czy punkt jest wyróżniony, jest obliczenie ilości zer na początku lub na końcu reprezentacji bitowej jednej ze współrzędnych punktu. Różny dobór tej cechy wpływa na pamięć oraz czas wymagany do znalezienia kolizji. Bardzo szeroki zakres punktów które spełniają kryteria bycia wyróżnionymi, spowoduje bardzo szybkie zapełnienie pamięci centralnego serwera a za wąski spowoduje, że czas do znalezienia kolizji się wydłuży.

2.3.2. Adding Walk

Adding Walk jest modyfikacją funkcji iteracyjnej f używanej w algorytmie Rho Pollard’a do obliczania kolejnych punktów na krzywej eliptycznej [31]. Funkcja ta opiera się na dodawaniu w każdej iteracji punktów z predefiniowanej tablicy punktów, co zapewnia wysoką efektywność oraz równomierny rozkład iteracji w grupie. Wprowadzenie Adding Walk, jak pokazano w pracy Teske [31], poprawia wydajność w stosunku do oryginalnej wersji algorytmu, zapewniając prostszą implementację w przypadku równoległych obliczeń, takich jak te wykonywane na GPU.

Niech $W_0 = nP$ będzie punktem startowym, gdzie n to znana wielokrotność generatora grupy P . Funkcja iteracyjna f jest zdefiniowana jako odwzorowanie $f : \langle P \rangle \rightarrow \{1, \dots, s\}$ o możliwie równomiernym rozkładzie. Następnie należy zdefiniować tablicę predefiniowanych punktów:

$$R_i = c_i P + d_i Q, \quad \text{dla } 0 \leq i \leq s-1,$$

gdzie c_i i d_i są współczynnikami losowymi. Funkcja iteracyjna jest zdefiniowana jako:

$$W_{i+1} = W_i + R_{f(W_i)}.$$

Podczas każdej iteracji konieczne jest zliczanie współczynników odpowiadających wielokrotnościom P i Q , aby każdy punkt na krzywej mógł zostać jednoznacznie przedstawiony w postaci $aP + bQ$. Suma współczynników a i b jest aktualizowana w każdej iteracji zgodnie z wartościami predefiniowanych współczynników c_i i d_i dla punktu $R_{f(W_i)}$. Zliczanie tych wartości jest kluczowe dla odtworzenia obliczeń w przypadku znalezienia kolizji, umożliwiając późniejsze wyznaczenie logarytmu dyskretnego.

Istotnym czynnikiem jest również rozmiar tablicy s , który ma spore znaczenie dla skuteczności algorytmu. Zbyt mała wartość s może powodować, że funkcja iteracyjna nie będzie wystarczająco "losowa". Eksperymenty wykazały, że dla $s \geq 16$, funkcja f zapewnia odpowiedni poziom losowości, niezależnie od rozmiaru grupy [31].

Główną zaletą tej metody w implementacjach GPU jest minimalizacja rozgałęzień w czasie iteracji. Dzięki temu niemal wszystkie wątki wykonują tę samą operację dodawania punktów, co jest istotne w architekturze SIMD. Wyjątkiem jest rzadki przypadek, gdy $W_i = R_{f(W_i)}$, co wymaga wykonania operacji zwielokrotnienia punktu.

3. Koncepcja

Projekt systemu do obliczania logarytmu dyskretnego wykorzystuje koprocesor GPU w celu akceleracji obliczeń. Równoległy algorytm Rho Pollard'a opiera się na architekturze, w której centralny serwer zbiera wyniki z wielu jednostek obliczeniowych. W związku z tym system składa się z dwóch głównych elementów: programu pełniącego rolę centralnego serwera oraz programu klienta, wykonującego obliczenia na GPU, który może być uruchamiany w wielu instancjach. Oba programy, zarówno serwer, jak i klient, mogą działać na jednym komputerze lub w środowisku maszyny wirtualnej, komunikując się za pomocą mechanizmów IPC. Architektura została zaprojektowana w sposób umożliwiający wykorzystanie wielu kart graficznych podłączonych do komputera, co dodatkowo zwiększa wydajność obliczeń.

3.1. Klient

Klient składa się z dwóch wyróżnionych części. Część odpowiedzialna za obliczenia na krzywej została napisana w języku CUDA C++. W celu optymalizacji obliczeń pod platformę GPU, algorytm oblicza kolejne punkty za pomocą *Adding Walk*, opisanego w poprzednim rozdziale. W tym celu niezbędne jest zaimplementowanie operacji modularnych, które umożliwiają przeprowadzanie obliczeń na krzywej eliptycznej.

Ponieważ w języku C oraz CUDA C++ nie istnieje typ danych pozwalający na przechowywanie liczb większych niż 64-bitowe, konieczne jest odpowiednie zaimplementowanie nowych typów danych oraz samych operacji na *długich liczbach*.

Aby rozpocząć obliczenia, program klienta otrzymuje następujące dane wejściowe:

- tablicę punktów startowych,
- ziarna użyte do wygenerowania każdego z punktów startowych,
- tablicę punktów wstępnie obliczonych potrzebnych do *Adding Walk*,
- parametry krzywej.

Ziarno używane do generowania punktów startowych jest w postaci liczby, przez którą należy z wielokrotnością punkt P , aby otrzymać dany punkt startowy. Po otrzymaniu danych, program uruchamia kernel CUDA, odpowiedzialny za szukanie punktów wyróżnionych. Każdy uruchomiony wątek, niezależnie od pozostałych, przeprowadza obliczenia algorytmu, aż do momentu gdy znajdzie punkt wyróżniony. Jako wynik działania, program zwraca tablicę punktów wyróżnionych, wraz z odpowiadającymi im ziarnami punktów startowych, od których zaczęły się obliczenia prowadzące do danego wyniku.

Druga część programu klienta, jest napisana w języku Python i stanowi wysokopoziomowy interfejs do komunikacji z serwerem. W jej skład wchodzi moduł odpowiedzialny za wywoływanie skompilowanej części kodu za pomocą ABI języka C, oraz kod funkcji, która jest uruchamiana jako oddzielny wątek z poziomu programu serwera. W celu komunikacji między wątkami wykorzystywana jest implementacja kolejki

asynchronicznej z biblioteki standardowej. Po uruchomieniu nowego wątku, na początku działania otrzymuje on parametry krzywej i pracuje on aż do znalezienia rozwiązania ECDLP i zakończenia wszystkich wątków przez serwer. W trakcie pracy, wątek klienta przyjmuje kolejne punkty startowe przesłane przez serwer za pomocą kolejki oraz zwraca obliczone punkty wyróżnione wraz z ziarnami.

3.2. Serwer

Serwer, w całości zaimplementowany w języku Python, pełni kluczową rolę w systemie, zajmując się gromadzeniem punktów wyróżnionych przesyłanych przez działające wątki klientów oraz wyszukiwaniem kolizji pomiędzy nimi. Jego zadania obejmują również zarządzanie pracą wątków klientów, polegające na dynamicznym uruchamianiu odpowiedniej liczby instancji oraz zapewnianiu mechanizmów komunikacji za pomocą kolejek. Do efektywnego przechowywania punktów wyróżnionych zastosowano strukturę danych w formie hash mapy, w której kluczami są współrzędne punktów. Rozwiązanie to umożliwia szybkie wyszukiwanie potencjalnych kolizji, co znacząco zwiększa wydajność procesu. Serwer również musi być zdolny do odtworzenia pełnego przebiegu obliczeń dowolnego klienta w przypadku, gdy jeden z jego punktów wyróżnionych stanie się elementem kolizji. Aby to osiągnąć, serwer implementuje tę samą logikę obliczeniową co klient, co zapewnia spójność wyników. Dzięki temu, na podstawie ziarna punktu startowego, serwer zawsze uzyskuje identyczne rezultaty jak klient. W przeciwieństwie do klienta, serwer dodatkowo śledzi sumę wielokrotności punktów P i Q , co jest niezbędne do późniejszego obliczenia logarytmu dyskretnego.

3.3. Przepływ działania

W momencie uruchomienia programu, serwer generuje wyznaczoną liczbę punktów wstępnie obliczonych. Punkty te będą przechowywane przez serwer aż do końca działania programu, wraz z parametrami, które wygenerowały każdy z tych punktów. Następnie serwer uruchamia w wielu wątkach funkcję napisaną w pythonie, która odpowiada za nadzorowanie pracy programu klienta GPU. Każda z uruchomionych funkcji w momencie startu otrzymuje adres dwóch kolejek, które będą służyły do komunikacji z głównym wątkiem serwera.

Po uruchomieniu wątków, serwer rozpoczyna działanie pętli, która zakończy się dopiero po znalezieniu rozwiązania ECDLP. W trakcie jej działania, serwer wykonuje następujące kroki:

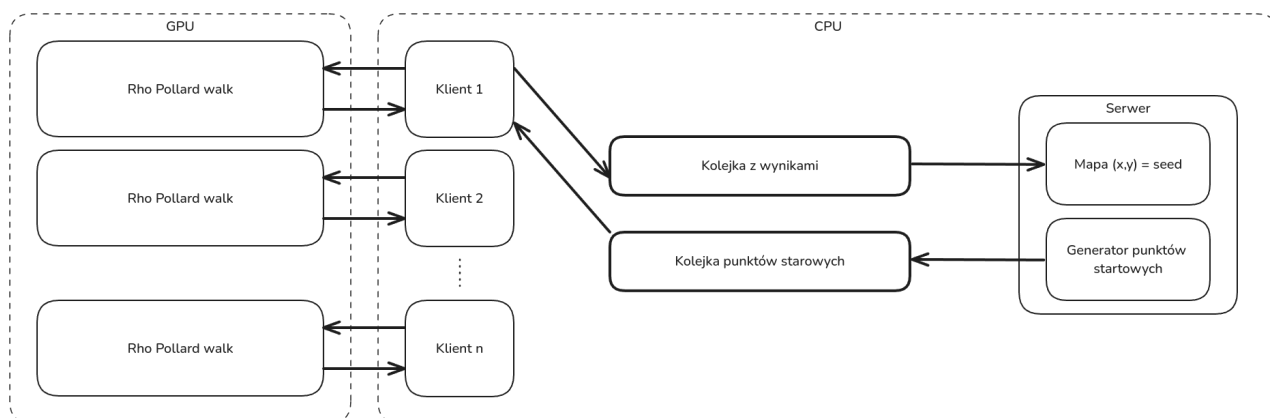
1. Generacja nowych punktów startowych
2. Przekazanie punktów startowych do kolejki zadań
3. Pobranie znalezionych punktów wyróżnionych z kolejki z wynikami
4. Sprawdzenie czy w bazie punktów znalezionych wystąpiła kolizja
5. Dodanie punktów do bazy punktów znalezionych

W momencie gdy serwer znajdzie kolizję dwóch punktów, podejmuje on próbę znalezienia ECDLP. Jeżeli wynik został poprawnie obliczony, serwer kończy działanie całego programu, wraz z wątkami klientów.

Podprogramy klientów, uruchomione w osobnych wątkach, również działają w pętli, w trakcie której wykonują następujące kroki:

1. Odebranie punktów startowych z kolejki zadań
2. Uruchomienie obliczeń na GPU
3. Odebranie wyników z GPU
4. Przekazanie znalezionych punktów wyróżnionych do kolejki z wynikami

Komunikacja z programem na GPU odbywa się za pomocą *Ctypes*, przekazując do programu GPU tablice z języka C oraz strukturę z danymi o parametrach krzywej, na której prowadzone są obliczenia.



Rys. 3.1. Schemat architektury

4. Implementacja

Ta część pracy jest poświęcona implementacji całego systemu, wraz ze szczegółowym opisem narzędzi oraz sprzętu wykorzystanego przy pracy nad projektem.

4.1. Oprogramowanie oraz platforma sprzętowa

4.1.1. Platforma sprzętowa

- OS: openSUSE Tumbleweed 20240812
- CPU: Intel Core i5-10600KF 4.10 GHz
- RAM: 16 GB
- GPU: GeForce RTX 2070 Super

4.1.2. Program po stronie CPU

Implementacja programu serwera działającego po stronie CPU, została wykonana z wykorzystaniem języka Python w wersji 3.11.1 wraz z pakietem obliczeniowym SageMath w wersji 10.4.beta3. Program klienta został zaimplementowany z wykorzystaniem języka Python w wersji 3.11.1 oraz języka CUDA C++ w wersji 12.4.

4.1.3. System budowania oraz kompilacja CUDA C++

W celu zarządzania kompilacją części projektu napisanej w języku CUDA oraz C++, zostało wykorzystane narzędzie *CMake*. *CMake* zapewnia natywne wsparcie dla języka CUDA, co znacznie upraszcza wszelkie zarządzanie kompilacją oraz linkowanie. Dodatkowo, wymaga to znacznie mniej wstępnej konfiguracji niż analogiczne rozwiązanie z wykorzystaniem samego narzędzia do budowania takiego jak *Make* lub *Ninja*.

Program napisany w CUDA kompilowany był z wykorzystaniem kompilatora NVCC dostarczonego wraz z pakietem *CUDA Toolkit* w wersji 12.4. NVCC do kompilacji kodu po stronie host'a (CPU) wykorzystuje kompilator z GCC w wersji 13.3.1.

4.1.4. Testy

Wszelkie testy poprawności implementacji rozwiązania są przeprowadzane z wykorzystaniem framework'a PyTest dla języka Python. Python wraz z wykorzystaniem pakietu obliczeniowego SageMath generuje testowe dane, które są przekazywane do programu działającego na GPU. Następnie, otrzymane wyniki są porównywane z rezultatami obliczonymi przy wykorzystaniu SageMath. Wykorzystanie PyTest oraz SageMath znacznie przyspieszyło pracę nad implementacją operacji na krzywej eliptycznej oraz docelowej implementacji algorytmu Rho Pollard'a. Możliwość szybkiego generowania dużych zbiorów danych testowych, pozwoliło na wczesne zauważenie wielu subtelnych błędów na etapie implementacji.

4.2. Program klienta Python

Część podprogramu klienta napisana w języku Python, została zaimplementowana za pomocą biblioteki *Threading* z biblioteki standardowej. Na początku działania programu, podprogram serwera uruchamia w kilku wątkach funkcję *GPUWorker* która stanowi implementację klienta. Odpowiada ona za dostosowywanie danych wejściowych do programu napisanego w CUDA, oraz zwracanie wyników z powrotem do serwera. Sygnatura funkcji GPUWorker:

```
1 def GPUworker(
2     starting_params: StartingParameters,
3     task_queue: Queue,
4     result_queue: Queue, stream):
```

W celu komunikacji z serwerem, wykorzystywane są obiekty kolejek *queue.Queue* z biblioteki standardowej. Ponieważ Python w wersji 3.12 nie wspiera równoległego wykonywania się wątków [28], to zmiana wątku następuje po wywołaniu programu CUDA. Następuje wtedy wywołanie systemowe *Sleep* i wątek pozostaje nieaktywny aż do zakończenia obliczeń przez GPU. Dzięki kompilacji programu CUDA z flagą *-default-stream per-thread*, każdy z wątków klienta uruchamia program na GPU w innym strumieniu CUDA. Pozwala to na współbieżne wykonywanie się obliczeń na GPU uruchomionych przez każdy wątek, zarządzane przez *CUDA Scheduler*.

Implementacja struktur danych przekazywanych do programu CUDA:

```
1 LIMBS = 5
2 class bn(ctypes.Structure):
3     _fields_ = [("array", ctypes.c_uint32 * LIMBS)]
4
5 class small_bn(ctypes.Structure):
6     _fields_ = [("array", ctypes.c_uint32 * ceil(LIMBS / 2))]
7 class EC_point(ctypes.Structure):
8     _fields_ = [
9         ("x", bn),
10        ("y", bn),
11        ("seed", bn),
12        ("is_distinguish", ctypes.c_uint32),
13    ]
```

4.3. Program klienta CUDA

Program uruchamiany na GPU został skompilowany jako dynamicznie linkowana biblioteka (*SHARED*). Aby zapewnić zgodność z interfejsem binarnym (ABI) wykorzystywanym przez bibliotekę Ctypes, do głównej funkcji dodano słowo kluczowe *extern "C"* (wydruk 4.1). Pozwoliło to uzyskać punkt wejścia zgodny z konwencją

wywołania C (C ABI). Główna funkcja, po otrzymaniu parametrów startowych, alokuje odpowiednią pamięć CUDA oraz konfiguruje niezbędne parametry uruchamiania dla głównego kernel'a, który realizuje algorytm Rho Pollard'a. Po zakończeniu obliczeń, pamięć jest zwalniana, a wyniki zapisywane w tablicach przekazanych jako dane wejściowe. W rezultacie, tablica punktów startowych przekazana do programu pełni jednocześnie funkcję miejsca zapisu wyników obliczeń.

```
1 extern "C" {  
2 void run_rho_pollard(  
3     EC_point *startingPts,  
4     uint32_t instances,  
5     uint32_t n, PCMP_point *precomputed_points,  
6     EC_parameters *parameters,  
7     int stream  
8 )
```

Wydruk 4.1. Prototyp funkcji wejściowej programu CUDA

Następne podrozdziały są poświęcone implementacji poszczególnych elementów składających się na kod głównego kernel'a CUDA.

4.4. Arytmetyka na ciele

4.4.1. Reprezentacja długich liczb

W przypadku ciała F_p , gdzie p jest liczbą pierwszą o rozmiarze 79 bit, potrzebny jest co najmniej 79 bitowy typ danych, do samego przechowywania liczb. Jednak nawet 79 bitowy typ danych nie wystarczy, aby przeprowadzić operację mnożenia dwóch liczb 79 bitowych. W takim przypadku, wynik pośredni może być maksymalnie $2 * 79 = 158$ bitowy. Dodatkowo, w celu reprezentacji takiej liczby, nie możemy się posłużyć wektorem składającym się z największego dostępnego natywnie typu danych, ponieważ wyniki pośrednie z operacji mnożenia lub dodawania mogą przekroczyć rozmiar słowa bitowego. W tym celu musimy wykorzystać typ danych mniejszy od maksymalnego. W przypadku CUDA C++, największy wspierany typ danych wynosi 64 bit, więc w celu reprezentacji liczb, w pracy wykorzystano wektor składający się z 32 bitowych słów. Najbliższa wielokrotność liczby 32 bitowej, większa niż 158 bit to 160 bit, dlatego w celu reprezentacji liczb na ciele, wykorzystywany jest wektor postaci:

$$\sum_{i=0}^4 x_i \cdot 2^{32i}$$

Zaimplementowany jako tablica typu `u_int32` (wydruk 4.2)

```
1 struct bn
2 {
3     uint32_t array[5];
4 };
```

Wydruk 4.2. Struktura do przechowywania długich liczb

Dla liczb, na których bezpośrednio nie będą wykonywane operacje arytmetyczne, wykorzystywany jest mniejszy, tymczasowy typ danych. Ograniczone zasoby szybkiej pamięci współdzielonej na GPU, wymuszają oszczędne zarządzanie pamięcią. W celu przechowywania wstępnie obliczonych punktów w pamięci współdzielonej, wykorzystywany jest mniejszy typ danych (wydruk 4.3)

```
1 struct small_bn
2 {
3     uint32_t array[3];
4 };
```

Wydruk 4.3. Mniejszy typ danych do długich liczb

Liczby w takiej postaci, przed przeprowadzeniem na nich operacji arytmetycznych, są ładowane z pamięci współdzielonej i z powrotem konwertowane na większy typ danych. Pozwala to zaoszczędzić $2 \cdot 32$ bitów na każdej liczbie, co w przypadku punktu składającego się z dwóch współrzędnych daje oszczędność $2 \cdot 2 \cdot 32 = 128$ bitów na punkt znajdujący się w pamięci.

4.4.2. Operacje na długich liczbach

Do operacji na długich liczbach, została odpowiednio dostosowana mała biblioteka dostępna w domenie publicznej: *tiny-bignum-c*. Biblioteka ta dostarcza podstawowe operacje na dużych liczbach w postaci wektorów, takie jak dodawanie, odejmowanie, mnożenie czy dzielenie. Wykorzystuje w tym celu standardowe algorytmy [23] wykorzystywane przy obliczeniach *multiple precision*. Dużą zaletą tej biblioteki jest jej prostota i brak wykorzystania standardowej biblioteki C oraz dynamicznej alokacji pamięci. Wszystkie operacje są wykonywane z wykorzystaniem stosu, co pozwala na jej wykorzystanie w środowiskach takich jak GPU, gdzie dostęp do dynamicznej alokacji pamięci jest ograniczony. W większości przypadków, aby dostosować kod z biblioteki do CUDA C++ wystarczyło dodanie odpowiedniej dyrektywy `__device__` przed każdą deklaracją funkcji, która informuje kompilator, że dana funkcja będzie wykonywana na GPU.

Prymitywy matematyczne dostarczane przez bibliotekę *tiny-bignum-c* nie są jednak wystarczające do przeprowadzenia operacji na ciele F_p . W związku z tym, zostały zaimplementowane dodatkowe operacje modularne takie jak mnożenie, dodawanie, odejmowanie i odwrotność modulo.

4.4.3. Dodawanie i odejmowanie modulo

Dodawanie oraz odejmowanie modulo p , wygląda bardzo podobnie do standardowego dodawania i odejmowania.

W przypadku odejmowania dwóch liczb na ciele F_p , nie ma potrzeby redukcji modulo p po każdej operacji. Jednak niezbędne jest upewnienie się, że wynik nie jest ujemny. Ponieważ prowadzone obliczenia są na liczbach bez znaku, to w sytuacji gdy $a - b = c$; $a < b$, nastąpi przepełnienie i wynik będzie w postaci $2^{32 \cdot n} - 1 - c$, gdzie n to rozmiar wektora do przechowywania liczb. Można bardzo łatwo wrócić do poprawnego wyniku wykorzystując jedną operację dodawania (wydruk 4.4) i korzystając z faktu, że wszystkie podstawowe operacje są wykonywane modulo $2^{32 \cdot n}$:

$$2^{32 \cdot n} - 1 - c + p \equiv p - c \pmod{2^{32 \cdot n}}$$

```
1 bignum_sub(a, b, c);
2 if (bignum_cmp(a, b) == SMALLER)
3 {
4     bignum_add(c, p, temp);
5     bignum_assign(c, temp);
6 }
```

Wydruk 4.4. Odejmowanie modulo

Dodawanie modulo p jest prostsze. Aby wykonać dodawanie modularne, wystarczy wykonać standardowe dodawanie i ewentualnie jeżeli $a + b \geq p$ zredukować wynik odejmując od niego liczbę p .

4.4.4. Mnożenie modulo

Mnożenie modularne jest bardziej kosztowne niż dodawanie czy odejmowanie, ponieważ wymaga dzielenia z resztą. W pracy przeprowadzane jest standardowe mnożenie modularne, jednak istnieją bardziej wydajne sposoby, na przykład wykorzystanie redukcji Barret'a [23].

Do przeprowadzenia klasycznego mnożenia modularnego na ciele F_p , należy na początku przeprowadzić standardowe mnożenie długich liczb. Następnie, otrzymany wynik jest dzielony przez p i zwracana jest reszta z dzielenia. Zarówno mnożenie jak i dzielenie z resztą, są funkcjami dostarczonymi w bibliotece *tiny-bignum-c*, więc mnożenie modularne sprowadza się do wykonania tych dwóch operacji jedna po drugiej.

4.4.5. Odwrotność modulo

Obliczanie odwrotności modulo p jest najbardziej kosztowną operacją na ciele F_p , głównie ze względu na wielokrotne dzielenie w pętli. Algorytm obliczania odwrotności modulo p został zaimplementowany z wykorzystaniem rozszerzonego algorytmu Euklidesa, który został zmodyfikowany do działania na liczbach nieujemnych. Główna

różnica względem klasycznego algorytmu, polega na wykorzystaniu dodatkowych zmiennych do śledzenia zmian znaku wyniku.

Prototypy wszystkich funkcji do operacji na długich liczbach, są widoczne na wydruku 4.5

Algorithm 1 Odwrotność modularna $a \bmod b$

```

1: Input:  $a, b$ 
2:  $b_0 \leftarrow b$ 
3:  $x_0 \leftarrow 0$ 
4:  $x_1 \leftarrow 1$ 
5:  $x_{0\_sign} \leftarrow 0$ 
6:  $x_{1\_sign} \leftarrow 0$ 
7: while  $a > 1$  do
8:    $q \leftarrow a \div b$ 
9:    $t \leftarrow b$ 
10:   $b \leftarrow a \bmod b$ 
11:   $a \leftarrow t$ 
12:   $t_2 \leftarrow x_0$ 
13:   $t_{2\_sign} \leftarrow x_{0\_sign}$ 
14:   $qx_0 \leftarrow q \times x_0$ 
15:  if  $x_{0\_sign} \neq x_{1\_sign}$  then
16:     $x_0 \leftarrow x_1 + qx_0$ 
17:     $x_{0\_sign} \leftarrow x_{1\_sign}$ 
18:  else
19:    if  $x_1 > qx_0$  then
20:       $x_0 \leftarrow x_1 - qx_0$ 
21:       $x_{0\_sign} \leftarrow x_{1\_sign}$ 
22:    else
23:       $x_0 \leftarrow qx_0 - x_1$ 
24:       $x_{0\_sign} \leftarrow 1 - x_{0\_sign}$ 
25:    end if
26:  end if
27:   $x_1 \leftarrow t_2$ 
28:   $x_{1\_sign} \leftarrow t_{2\_sign}$ 
29: end while
30: if  $x_{1\_sign} == 1$  then
31:   return  $b - x_1$ 
32: else
33:   return  $x_1$ 
34: end if

```

```
1 __device__ void bignum_init(struct bn *n);
2 __device__ void bignum_from_int(struct bn *n, DTYPE_TMP i);
3 __device__ int bignum_to_int(struct bn *n);
4
5 __device__ void bignum_add(struct bn *a, struct bn *b, struct bn *c);
6 __device__ void bignum_sub(struct bn *a, struct bn *b, struct bn *c);
7 __device__ void bignum_mul(struct bn *a, struct bn *b, struct bn *c);
8 __device__ void bignum_div(struct bn *a, struct bn *b, struct bn *c);
9 __device__ void bignum_mod(struct bn *a, struct bn *b, struct bn *c);
10 __device__ void bignum_divmod(
11     struct bn *a, struct bn *b, struct bn *c, struct bn *d
12 );
13
14 __device__ void bignum_assign_fsmall(
15     struct bn *dst, struct small_bn *src
16 );
17 __device__ void bignum_assign_small(
18     struct small_bn *dst, struct small_bn *src
19 );
20
21 __device__ void bignum_modinv(struct bn *a, struct bn *b, struct bn *c);
22
23 __device__ void bignum_and(struct bn *a, struct bn *b, struct bn *c);
24 __device__ void bignum_or(struct bn *a, struct bn *b, struct bn *c);
25 __device__ void bignum_xor(struct bn *a, struct bn *b, struct bn *c);
26 __device__ void bignum_lshift(struct bn *a, struct bn *b, int nbits);
27 __device__ void bignum_rshift(struct bn *a, struct bn *b, int nbits);
28
29 __device__ int bignum_cmp(struct bn *a, struct bn *b);
30 __device__ int bignum_is_zero(struct bn *n);
31 __device__ void bignum_inc(struct bn *n);
32 __device__ void bignum_dec(struct bn *n);
33 __device__ void bignum_assign(struct bn *dst, struct bn *src);
```

Wydruk 4.5. Prototypy funkcji wykorzystywanych do operacji na długich liczbach

4.4.6. Biblioteka CGBN

W pierwotnej wersji pracy, do operacji na dużych liczbach, była wykorzystywana specjalna biblioteka CGBN dla platformy CUDA. Oferuje ona wszystkie podstawowe operacje na długich liczbach, takie jak dodawanie, odejmowanie czy mnożenie nawet do 32 tys. bitów. Dodatkowo, posiada ona implementację bardziej zaawansowanych funkcji, takich jak odwrotność modulo czy redukcja Barret’a. Niestety, biblioteka ta narzuca spore ograniczenia pod kątem zasobów. Niezbędne jest grupowanie wątków w grupy 4, 8, 16 lub 32. Wy-

soką wydajność obliczeń, udało się uzyskać dopiero w grupach składających się z 32 wątków. Pomimo znacznie szybszego wykonywania się poszczególnych operacji w ramach takiej grupy wątków, narzucone ograniczenia oraz wysokie użycie rejestrów uniemożliwiało efektywne zaimplementowanie dużej ilości takich instancji działających równolegle. Całkowita wydajność mierzona w ilości operacji na krzywej na sekundę osiągnięta z jej wykorzystaniem, była średnio 4.57 razy gorsza niż z wykorzystaniem znacznie prostszej implementacji na bazie *tiny-bignum-c*.

4.5. Funkcja iterująca

Główna funkcja realizująca kolejne kroki algorytmu Rho Pollard'a została zaimplementowana jako kernel CUDA.

4.5.1. Rozpoczęcie obliczeń

Na początku swojego działania kernel ładuje punkty wstępnie obliczone do pamięci współdzielonej (*shared memory*), która pełni funkcję pamięci podręcznej (wydruk 4.6).

Pierwszy wątek każdego bloku inicjalizuje specjalną flagę `warp_finished`, ustawiając jej wartość na 0. Flaga ta służy do kontrolowania zakończenia pracy wszystkich wątków w ramach danego bloku. Po zakończeniu obliczeń przez przynajmniej jeden wątek, flaga sygnalizuje konieczność zakończenia działania pozostałych wątków w bloku. Szczegółowe omówienie mechanizmu działania tej flagi znajduje się w sekcji poświęconej (*tail effect*).

```

1  if (threadIdx.x == 0)
2  {
3      printf("STREAM %d BLOCK %d started\n", stream, blockIdx.x);
4      for (int i = 0; i < PRECOMPUTED_POINTS; i++)
5      {
6          bignum_assign_small(&SMEMprecomputed[i].x, &args.precomputed[i].x);
7          bignum_assign_small(&SMEMprecomputed[i].y, &args.precomputed[i].y);
8      }
9      warp_finished = 0;
10 }
11 __syncthreads();

```

Wydruk 4.6. Inicjalizacja pamięci współdzielonej i flagi `warp_finished`

4.5.2. Wstępnie obliczone punkty

Dostęp do wstępnie obliczonych punktów jest niezbędny w każdej iteracji *Addition walk.*, dlatego ważne jest, aby przechowywać je w szybkiej pamięci. Wykorzystałem w tym celu pamięć współdzieloną *shared memory*. W przeciwieństwie do znacznie większej pamięci globalnej, jest ona przechowywana bezpośrednio na SM [11, 19]. Ponieważ liczba punktów która zapewnia dostatecznie losowy spacer po krzywej eliptycznej jest stosun-

kowo niewielka [31], to rozmiar pamięci jest wystarczający. W docelowej wersji, przechowywane jest 128 punktów.

Funkcja przydzielająca punkt wstępnie obliczony, na podstawie aktualnie sumowanego punktu została zaimplementowana poprzez operację AND maski bitowej z pierwszymi 64 bitami współrzędnej x punktu. W ten sposób, każdy punkt W_i jest przyporządkowany do jednego z 128 wstępnie obliczonych punktów, a następnie punkty są dodawane do siebie.

```
1 #define PRECOMPUTED_POINTS 128
2 __device__ uint32_t map_to_index(bn *x) {
3     return (x->array[0] & (PRECOMPUTED_POINTS - 1));
4 }
```

Wydruk 4.7. Funkcja przydzielająca punkt

4.5.3. Punkty wyróżnione

W ramach każdej iteracji, należy w wydajny i szybki sposób sprawdzić, czy obliczony punkt jest punktem wyróżnionym. W tej implementacji, kryterium warunkującym jest liczba zer na końcu współrzędnej x obliczonego punktu.

Do sprawdzenia, czy punkt jest wyróżniony, służy prosta funkcja obliczająca bitową operację AND ze współrzędnej punktu oraz specjalnej maski bitowej wyznaczonej na podstawie poszukiwanej ilości zer. Przykładowo, dla poszukiwanej liczby zer 3, maska będzie w postaci 0...00111. Jeżeli również ostatnie 3 bity współrzędnej x będzie miało zerowy znak, to otrzymany wynik operacji AND wyniesie 0.

Istotne jest, aby taka sama funkcja została zaimplementowana po stronie serwera na CPU, ponieważ w przypadku znalezienia kolizji, musi on być w stanie odtworzyć cały spacer losowy prowadzący do danego punktu wyróżnionego.

Algorithm 2 Funkcja `is_distinguish`

```
1: Input:  $x$ ,  $liczba\_zer$ 
2:  $mask \leftarrow 1 \ll liczba\_zer - 1$ 
3: if  $(x \& mask) == 0$  then
4:     return true
5: else
6:     return false
7: end if
```

Aby ułatwić testy na różnych etapach implementacji całego systemu, liczba sprawdzanej ilości zer jest sparametryzowana. Przy starcie każdej serii obliczeń, poszukiwana liczba zer jest jednym z parametrów przekazywanym do funkcji kernel'a.

Po znalezieniu punktu wyróżnionego, ustawiana jest specjalna flaga w strukturze przechowującej dane punktu oraz jest on zapisywany pod pierwsze wolne miejsce w pamięci globalnej.

4.5.4. Obliczanie odwrotności w seriach

Każda operacja dodawania punktów na krzywej eliptycznej we współrzędnych afinicznych, składa się z 3 mnożeń modularnych (M) oraz jednej operacji obliczania odwrotności w ciele (O). Przeprowadza się również operację dodawania i odejmowania modularnego, jednak ich koszt obliczeniowy jest pomijalnie mały [5].

Koszt dodawania punktów na krzywej:

$$1O + 3M$$

Działaniem, które wpływa na wysoki koszt obliczeniowy mnożenia modularnego oraz obliczania odwrotności w ciele, jest operacja dzielenia [23]. Przykładowo, wykorzystywana w pracy implementacja dzielenia stosuje prosty algorytm *long division*, o złożoności obliczeniowej $O(n^2)$.

Warto zauważyć, że operacja obliczania odwrotności modularnej z wykorzystaniem algorytmu euklidesa, wykonuje dzielenie podczas obliczania modulo, na każdym etapie pętli (algorytm 1). To czyni ją najkosztowniejszą obliczeniowo operacją na ciele, znacznie kosztowniejszą niż operacja mnożenia modularnego.

Aby przyspieszyć obliczenia, w pracy zastosowano technikę obliczania wielu odwrotności równocześnie, znaną jako *Montgomery Trick* [24].

Niech x_1, \dots, x_n będą elementami, dla których należy policzyć odwrotność. Na początku obliczana jest tablica elementów, w postaci $a_1 = x_1, a_2 = x_1 \cdot x_2, \dots, a_n = x_1 \cdot \dots \cdot x_n$. Następnie, obliczana jest odwrotność ostatniego elementu a_n za pomocą jednej operacji odwrotności w ciele. Teraz, aby policzyć odwrotność elementu x_n wystarczy wykonać jedynie operację mnożenia $b_n = a_{n-1} \cdot a_n^{-1}$. Kolejne elementy są obliczane analogicznie, za pomocą mnożenia: $b_{n-1} = a_{n-2} \cdot b_n$. *Montgomery trick* pozwala na zamianę:

$$nO = O + 3(n - 1)M$$

Sposób implementacji tej metody wymagał podjęcia paru decyzji. Pierwszym problemem który pojawia się w metodzie Montgomery'ego, jest konieczność obliczania iteracji dla wielu punktów jednocześnie.

Jednym ze sposobów by tego dokonać, jest zsynchronizowanie wielu wątków w ramach bloku obliczeniowego, a następnie przekazanie jednemu z nich, za pomocą pamięci współdzielonej, wszystkich liczb do obliczenia odwrotności. Następnie, wątek zwracałby obliczone odwrotności za pomocą pamięci współdzielonej. Jak zauważono w pracy [7], takie podejście nie jest optymalne w przypadku GPU. Wymagałoby to sporo synchronizacji pomiędzy wątkami, oraz sporej ilości zapisów i odczytów z pamięci współdzielonej, która pomimo bycia znacznie szybszą niż pamięć globalna, nie jest tak szybka jak prywatna pamięć w postaci rejestrów. Dodatkowo, z racji, że tylko jeden wątek oblicza odwrotności, pozostałe muszą beczynnie czekać.

Dlatego sposobem, który został zastosowany jest obliczanie wielu odwrotności w ramach jednego wątku. Oznacza to, że każdy wątek zamiast przetwarzać tylko jeden punkt startowy, dostaje ich n na początku działania programu.

Naiwne podejście polegałoby na przekazaniu każdemu z wątków n punktów startowych, i oczekiwanie aż znajdzie dla każdego punktu startowego odpowiadający mu punkt wyróżniony. Taki sposób powoduje, że bardzo szybko zysk z metody Montgomery’ego jest tracony, a nawet zaczyna pojawiać się dodatkowy koszt obliczeniowy, z powodu obliczeń, których nie wykorzystujemy. Wynika to z faktu, że po znalezieniu i punktów, zysk z metody jest postaci:

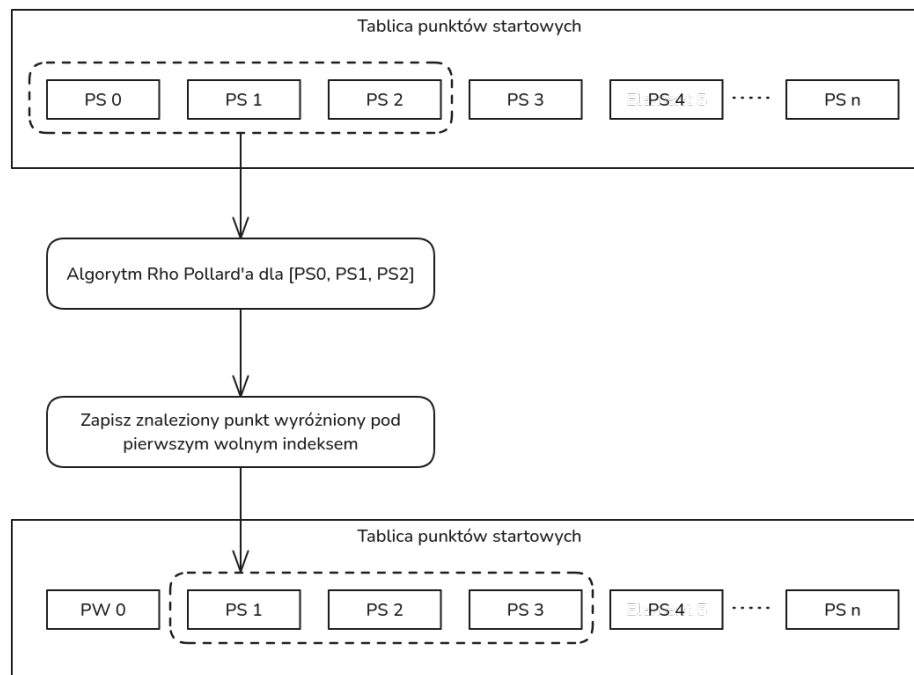
$$(n - i)O = O + 3(n - i)M + 3(i)M$$

Gdzie $3(i)M$ to mnożenia, które nie przyczyniają się do znalezienia nowych punktów wyróżnionych, więc należy je traktować jako niepotrzebne spowolnienie.

W celu rozwiązania tego problemu, w pracy zostało zastosowanie okienkowanie obliczeń w tablicy o stałym rozmiarze m . Sposób ten wymaga, aby każdy wątek otrzymał $n > m$ punktów startowych w wyznaczonym dla niego miejscu w pamięci globalnej. Im większa różnica $n - m$ tym lepszy stosunek czasu obliczeń do narzutu czasowego związanego ze startem programu.

Na początku działania wątku, ładowane jest do tablicy okna kolejne m punktów startowych z pamięci globalnej. Następnie, w pętli obliczane są odwrotności wymagane dla operacji dodawania punktów i przeprowadzane jest ich dodawanie w ramach algorytmu Rho Pollard’a. Tym sposobem, w jednym kroku pętli, wykonywana jest jedna iterację algorytmu Rho Pollard’a dla m punktów. Na samym końcu każdego kroku pętli, jest sprawdzane czy któryś z obliczonych punktów jest punktem wyróżnionym. Jeżeli tak, znaleziony punkt zapisywany jest na pierwszym wolnym miejscu w pamięci globalnej, a na jego miejsce do tablicy okna ładowany jest kolejny punkt startowy. Dzięki temu, przez większość czasu obliczeń, wykorzystywany jest pełny zysk z metody Montgomery’ego. Przykład dla $m = 3$ ilustrujący działanie okna, znajdują się na rys. 4.1. Punkty startowe zostały przedstawione jako *PS*, znaleziony punkt wyróżniony jako *PW*.

Jeżeli dodatkowo zastosujemy flagę, która kończy obliczenia wszystkich wątków w bloku, gdy pierwszy wątek, znajdzie $n - m$ punktów wyróżnionych, to otrzymujemy maksymalny poziom wydajności w ciągu całej fali obliczeń. Efekt ten jest dokładniej opisany w części poświęconej problemowi *tail effect*.



Rys. 4.1. Schemat działania jednego kroku pętli z wykorzystaniem okna

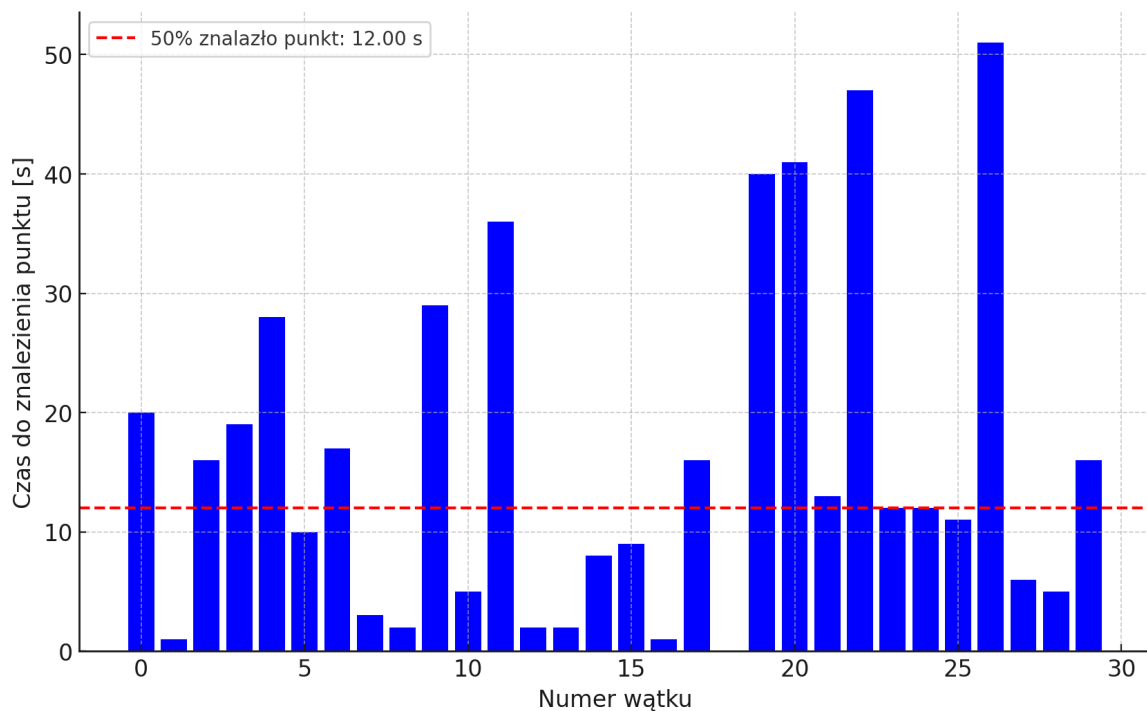
4.6. Tail effect

Tail Effect to zjawisko, które polega na nierównomiernym obciążeniu wątków na GPU. W przypadku równoległej wersji algorytmu Rho Pollard’a, to zjawisko jest szczególnie widoczne, ze względu na losowość funkcji błędzenia po krzywej. Wątki znajdują punkty wyróżnione w różnym, losowym czasie.

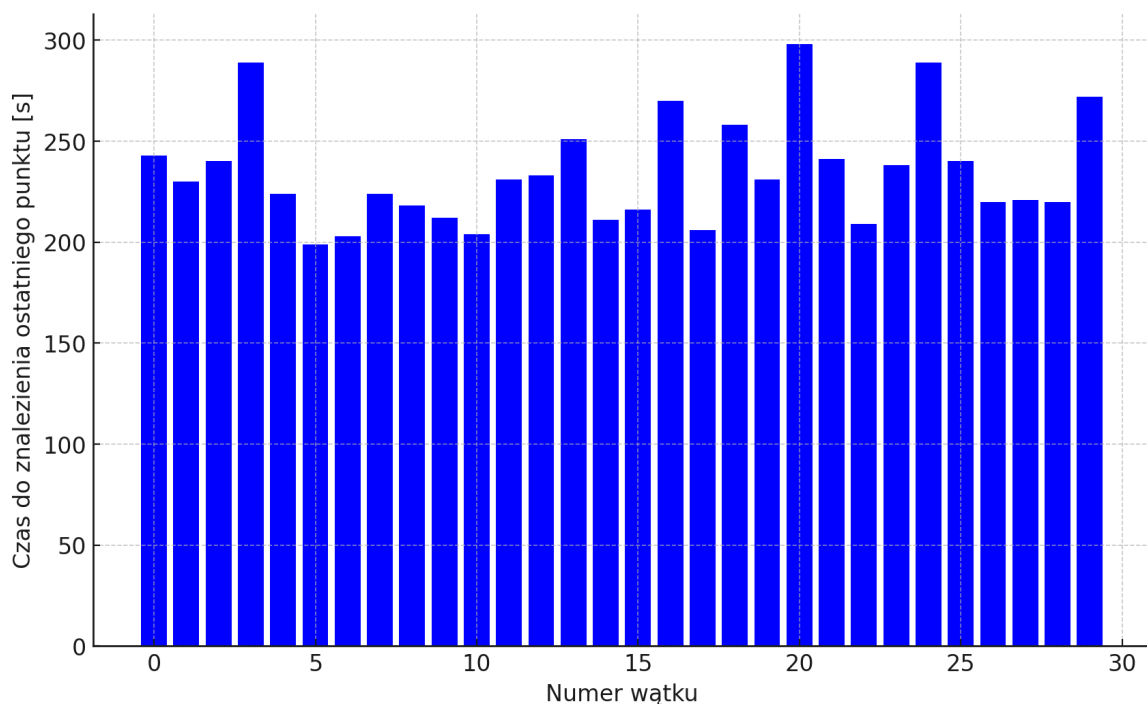
Po znalezieniu punktu wyróżnionego, wątek musi poczekać, na zakończenie pracy wszystkich pozostałych wątków w ramach jednego bloku. To powoduje, że tylko przez krótki czas są wykorzystywane wszystkie zasoby GPU.

Jednym ze sposobów na zmniejszenie tego efektu, jest wydłużenie czasu pracy każdego z wątków poprzez zwiększenie liczby punktów wyróżnionych, które musi znaleźć. Dzięki temu spada prawdopodobieństwo, że zakończy on swoją pracę znacznie wcześniej niż pozostałe wątki w bloku. Zastosowanie metody z oknem, opisanej w poprzedniej sekcji, pozwala na znacznie lepszą utylizację zasobów przez większość czasu obliczeń.

Rys 4.2 oraz 4.3 przedstawiają czas pracy każdego z wątków w bloku, w trakcie jednej fali obliczeń. W trakcie testu, każdy z wątków poszukiwał punktów wyróżnionych z 17 zerami na końcu współrzędnej x . Widoczne jest znacznie lepsze wykorzystanie zasobów GPU w przypadku, gdy każdy z wątków musi znaleźć 3 punkty wyróżnione przed zakończeniem pracy (rys. 4.3). Zmniejsza to wpływ anomalii, gdy punkt wyróżniony jest znajdowany znacznie wcześniej lub później niż wynika z wartości oczekiwanej.



Rys. 4.2. Tail effect. Wątki kończą pracę po znalezieniu jednego punktu wyróżnionego.



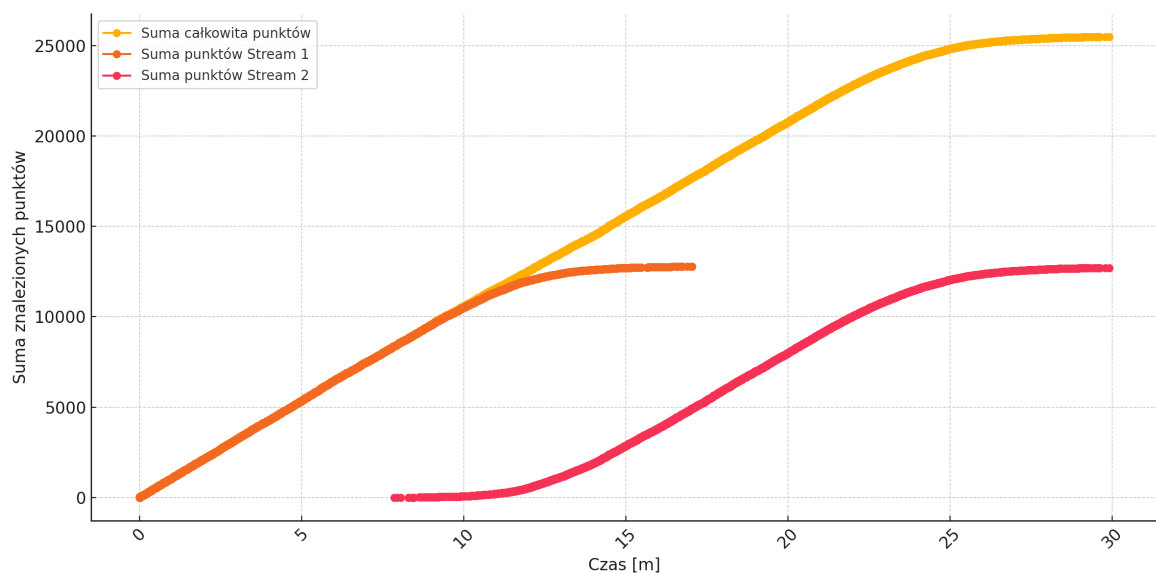
Rys. 4.3. Tail effect. Wątki kończą pracę po znalezieniu 3 punktów wyróżnionych.

Aby jeszcze bardziej zniwelować efekt, została zastosowana specjalna flaga w pamięci współdzielonej, która pozwala zakończyć obliczenia wszystkich wątków w bloku, po tym, jak pierwszy z nich znajdzie wszystkie punkty wyróżnione. Pozwala to zwolnić miejsce

na kolejny blok i pozwolić na załadowanie nowych danych do obliczeń, gdy tylko poziom wykorzystania zasobów GPU zacznie spadać. Karty graficzne z architekturą Turing, pozwalają na kolejgowanie wielu uruchomień kernel'a wykorzystując mechanizm strumieni CUDA. Dzięki wykorzystaniu strumieni, scheduler CUDA może uruchamiać kolejne bloki obliczeń, gdy tylko zwolni się miejsce na kolejny blok, zamiast czekać ze startem na zakończenie poprzedniego kernel'a [19].

Rys 4.4 przedstawia wykres sumy znalezionych punktów od czasu przy uruchomieniu dwóch kernel'i w oddzielnych strumieniach. Każdy z wątków szukał 10 punktów wyróżnionych z 22 zerami na końcu współrzędnej x . Zastosowano flagę kończenia obliczeń wszystkich wątków w bloku.

Widoczne jest przejmowanie przez strumień Stream 2 wolnych zasobów, zwalnianych przez Stream 1. Wyplaszczenie krzywej sumy punktów pod koniec pracy każdego ze strumieni, jest efektem ogona, spowodowanym przez nierównomierny czas pracy na poziomie bloków. Jednak dopóki jest zapewniona zastępowalność bloków na SM, efekt nie wpływa na wydajność obliczeń.



Rys. 4.4. Suma znalezionych punktów wyróżnionych z podziałem na strumienie.

4.7. Program serwera

Główny cel serwera centralnego w równoległej wersji algorytmu Rho Pollard'a, sprowadza się do przechowywania obliczonych punktów wyróżnionych oraz poszukiwania wśród nich kolizji. W tej pracy, jego zadania zostały rozszerzone o generowanie punktów startowych oraz wstępną generację punktów niezbędnych do spaceru losowego w wersji *addition walk*.

Na samym początku działania systemu, serwer startuje odpowiednią ilość klientów, poprzez uruchamianie w osobnych wątkach funkcji *GPUWorker*, w ilości na tyle dużej,

aby wysycić zasoby GPU. Ponieważ kod działający na GPU został skompilowany z flagą dla kompilatora NVCC `-default-stream per-thread`, każdy z klientów działających w osobnym wątku, tworzy własny strumień CUDA, co pozwala na kolejkowanie uruchomionych kernel'i po stronie GPU.

Klienci komunikuje się z serwerem za pomocą dwóch asynchronicznych kolejek FIFO. Pierwsza z nich służy do przekazywania punktów startowych z serwera do klientów, a druga do przekazywania znalezionych punktów wyróżnionych przez klientów do serwera. Taka centralizacja generowania punktów startowych, wynika z problemów z bibliotekami SageMath, podczas uruchamiania ich kodu w wielu wątkach jednocześnie. Serwer w celu przechowywania punktów otrzymanych od klientów, wykorzystuje zwykły słownik dostępny w języku Python, który jest odpowiednikiem hash-mapy. Punkty są przechowywane w formie: $(x,y): seed$, gdzie *seed* oznacza ziarno z jakiego został wygenerowany punkt startowy, który doprowadził do znalezienia punktu wyróżnionego. Dzięki temu, w razie kolizji, serwer jest w stanie odtworzyć punkt startowy, a następnie wykonać cały spacer losowy, który doprowadził do danego punktu. Jest to szczególnie istotne, ponieważ po stronie GPU nie są zliczane parametry a oraz b , niezbędne do obliczenia logarytmu dyskretnego

4.7.1. Generacja punktów startowych

Początkowo, punkty startowe były generowane za pomocą funkcji skrótu MD5 z operacją modulo rzędu ciała, jednak bardziej wydajnym podejściem okazało się wykorzystanie pakietu `random` z biblioteki standardowej języka Python. Pakiet ten zapewnia wystarczającą losowość do tego zastosowania, co umożliwia szybszą i efektywną generację punktów.

Proces generacji punktów startowych odbywa się wyłącznie po stronie serwera (wydruk 4.8). Wygenerowane punkty, wraz z odpowiadającymi im ziarnami (*seed*), są przekazywane do klienta. Jeśli jednak wygenerowany punkt startowy spełnia od razu kryteria punktu wyróżnionego, zostaje przekazany bezpośrednio do puli punktów wyróżnionych i nie zajmuje miejsca wśród punktów startowych (linia 14). Dzięki temu zasoby są lepiej wykorzystywane, a redundancja w obliczeniach jest zredukowana.

```

1 def generate_starting_points(instances, zeros_count):
2     distinguish_points = []
3     starting_points = []
4     i = 0
5     while i < instances:
6         seed = int.from_bytes(random.randbytes(10), "big") % curve_order
7         A = P * seed
8         x = int(A[0])
9         y = int(A[1])
10        if not is_distinguish(x, zeros_count):
11            i += 1
12            starting_points.append((x, y, seed))
13        else:
14            distinguish_points.append((x, y, seed))
15    return starting_points, distinguish_points

```

Wydruk 4.8. Generacja punktów startowych

4.7.2. Kolizje

W przypadku wykrycia kolizji serwer pobiera ziarna obu punktów i odtwarza kolejne kroki algorytmu, które doprowadziły do ich znalezienia. W tym procesie serwer oblicza wielokrotności punktów P oraz Q , co pozwala na określenie logarytmu dyskretnego. W rzadkich przypadkach, gdy obliczenie odwrotności modularnej jest niemożliwe (np. mianownik wynosi 0), obliczenie logarytmu dyskretnego może się nie powieść.

Jeżeli jednak obliczenia zakończą się sukcesem, serwer wyznacza logarytm dyskretny i wysyła sygnał do zakończenia dalszych obliczeń. Wynik końcowy jest następnie przekazywany na standardowe wyjście (stdout).

Implementacja funkcji obliczającej współczynniki a i b w Pythonie:

```
1 def calculate_ab(seed, precomputed_points: list[PrecomputedPoint]):
2     a_sum = seed
3     b_sum = 0
4     W = P * seed
5     while not is_distinguish(W[0], ZEROS_COUNT):
6         precomp_index = map_to_index(W[0])
7         precomputed = precomputed_points[precomp_index]
8         R = precomputed.point
9         a_sum = a_sum + precomputed.a
10        b_sum = b_sum + precomputed.b
11        W = W + R
12    a_sum = a_sum % curve_order
13    b_sum = b_sum % curve_order
14    return (a_sum, b_sum)
```

Wydruk 4.9. Obliczanie współczynników a i b

Funkcja `calculate_ab` przyjmuje ziarno oraz listę wcześniej obliczonych punktów. Kolejne wielokrotności punktów P i Q są sumowane wraz z ich współczynnikami a i b , aż do napotkania punktu wyróżnionego. Wynikowe wartości są redukowane modulo rząd krzywej i zwracane jako współczynniki, które umożliwiają wyznaczenie logarytmu dyskretnego.

4.8. Logowanie

W celu monitorowania wydajności systemu oraz postępu obliczeń, zarówno w kodzie klienta, jak i serwera, zaimplementowano mechanizm logowania. CUDA umożliwia wypisywanie informacji na standardowe wyjście (`stdout`) za pomocą funkcji `printf`, analogicznie jak w standardowym języku C. W kodzie głównego kernel'a można aktywować logowanie, ustawiając odpowiednią flagę za pomocą dyrektywy `#define logging`.

Logowane dane w kernel'u obejmują:

- czas znalezienia punktu wyróżnionego,
- numer wątku,
- numer bloku,
- numer strumienia.

Przykładowy fragment logów generowanych przez kernel:

```
Wed Sep 18 01:14:33 AM CEST 2024 STREAM 0, instance: 8253 found distinguish point 0
Wed Sep 18 01:14:33 AM CEST 2024 STREAM 0, instance: 28320 found distinguish point 0
Wed Sep 18 01:14:33 AM CEST 2024 STREAM 0, instance: 19113 found distinguish point 0
Wed Sep 18 01:14:33 AM CEST 2024 STREAM 0, instance: 28214 found distinguish point 0
Wed Sep 18 01:14:33 AM CEST 2024 STREAM 0, instance: 24343 found distinguish point 0
```

Dodatkowo, serwer po każdorazowym otrzymaniu danych od jednego z wątków klienta, wypisuje informacje o łącznej liczbie punktów wyróżnionych które zgromadził do tej pory. Przykładowe logi generowane przez serwer:

```
Wed Sep 18 07:13:45 PM CEST 2024 Got new distinguish points
Wed Sep 18 07:13:45 PM CEST 2024 Currently have 941239
Wed Sep 18 07:13:45 PM CEST 2024 GPU worker 0 got task
Wed Sep 18 07:13:45 PM CEST 2024 Got 957733 points
```

Mechanizm logowania umożliwił dokładniejszą analizę wykorzystania zasobów GPU oraz optymalne dostosowanie liczby wątków i bloków do struktury sprzętowej. Dane logowane były zapisywane do pliku, co pozwoliło na dalsze przetwarzanie, takie jak mierzenie wydajności i wizualizację wyników w formie wykresów. Wizualizacje te przedstawiają między innymi sumę znalezionych punktów od czasu, co pozwoliło oszacować wydajność projektu.

5. Wyniki

5.1. Testy implementacji

Głównym celem pracy było znalezienie rozwiązania problemu logarytmu dyskretnego dla krzywej ECCp79 z Certicom Challenge w możliwie najkrótszym czasie. Mimo że rozmiar krzywej jest stosunkowo niewielki, a współczesny sprzęt znacznie wydajniejszy od tego dostępnego w momencie pierwszych prób złamania tej krzywej, obliczenie rozwiązania wciąż zajmuje czas liczony w godzinach.

Ze względu na czasochłonność pełnych obliczeń, niezbędne było przeprowadzenie testów implementacji na mniejszych, uproszczonych problemach, które pozwoliłyby na szybszą weryfikację poprawności algorytmu i jego działania. Testy te były niezbędnym krokiem przed przystąpieniem do pełnoprawnych prób rozwiązania problemu logarytmu dyskretnego dla krzywej ECCp79. Pozwoliły one na upewnienie się, że implementacja działa poprawnie oraz spełnia wszystkie założenia teoretyczne.

W tym celu zostały wykorzystane środowiska PyTest oraz SageMath do przeprowadzenia testów funkcjonalnych, weryfikujących poprawność operacji na krzywych eliptycznych oraz kolejnych kroków algorytmu Rho Pollard'a.

Testy obejmowały następujące aspekty:

- Poprawność dodawania punktów na krzywej eliptycznej: Wyniki obliczeń programu klienta były porównywane z wynikami uzyskanymi w SageMath.
- Weryfikacja pierwszych kilkuset iteracji algorytmu Rho Pollard'a: Sprawdzano zgodność wyników generowanych przez implementację z oczekiwanymi wartościami teoretycznymi.
- Sprawdzenie integralności przesyłanych danych pomiędzy serwerem a klientem: Testowano, czy format danych oraz zawartość są poprawnie przetwarzane na styku kodu Python'a oraz CUDA.

Wszystkie testy zostały pomyślnie zaliczone w obecnej wersji programu, co potwierdza poprawność implementacji oraz zgodność wyników z teoretycznymi oczekiwaniami.

5.2. Wydajność

W celu oceny wydajności implementacji program był wielokrotnie uruchamiany z włączonym pełnym logowaniem każdego znalezionego punktu wyróżnionego. Wydajność mierzono na podstawie przyrostu liczby znalezionych punktów w czasie, co pozwalało na oszacowanie liczby iteracji dodawania punktów na sekundę.

Testy zostały przeprowadzone przy założeniu, że punkt wyróżniony spełnia warunek 20 najmłodszych bitów współrzędnej x równych 0 w reprezentacji bitowej. Sumę znalezionych punktów wyróżnionych została obliczona w różnych odstępach czasu: po 1 minucie oraz 5 minutach.

Tabela 5.1. Wyniki testów wydajności

Czas testu	Liczba znalezionych punktów	Średnia liczba iteracji/s
1 minuta	5274	87,9 mln
5 minut	26310	87,7 mln

Na podstawie wyników testów oszacowano liczbę operacji dodawania punktów na sekundę w algorytmie Rho Pollard’a. Prawdopodobieństwo znalezienia punktu wyróżnionego, gdy 20 najmłodszych bitów współrzędnej x wynosi 0, można określić jako

$$P = 2^{-20}.$$

Średnia liczba iteracji potrzebna do znalezienia punktu wyróżnionego wynosi więc:

$$\text{Średnia liczba iteracji} = \frac{1}{P} = 2^{20}.$$

Na podstawie wyników testów, można stwierdzić, że implementacja osiąga wydajność na poziomie 87,7 milionów operacji dodawania punktów na sekundę.

5.3. Rozwiązanie ECDLP dla krzywej ECCp79

Najważniejszym wyznacznikiem osiągnięcia założonego celu tej pracy było znalezienie poprawnego rozwiązania ECDLP dla krzywej z challenge’u Certicom ECCp79. Po zweryfikowaniu poprawności implementacji na znacznie mniejszych rozmiarach krzywych eliptycznych, przystąpiono do przeprowadzenia obliczeń dla docelowego problemu. Proces ten wymagał wykonania dużej liczby iteracji algorytmu Rho Pollard’a, a jego wyniki zostały szczegółowo udokumentowane w postaci zapisów logów.

```

Wed Sep 18 09:58:58 PM CEST 2024 Got new distinguish points
Wed Sep 18 09:58:58 PM CEST 2024 Currently have 1856364
Wed Sep 18 09:58:58 PM CEST 2024 Collision!GPU worker 19 got task
Wed Sep 18 09:58:58 PM CEST 2024
Wed Sep 18 09:58:58 PM CEST 2024 (383565681993649705975808 : 104190497174372507008664 : 1)
Wed Sep 18 09:58:58 PM CEST 2024 Seed 1: 441886239795098360035723
Wed Sep 18 09:58:58 PM CEST 2024 Seed 2: 98086214830357275751475
Wed Sep 18 09:58:59 PM CEST 2024 GPU worker 19 started processing
Wed Sep 18 09:58:59 PM CEST 2024 Starting rho pollard: zeroes count 20Launched rho pollard stream 19
Wed Sep 18 09:59:10 PM CEST 2024 STREAM 3 BLOCK 9 started
Wed Sep 18 09:59:55 PM CEST 2024 a1: 121214231505221642106197, b1: 239868006679437883471744
Wed Sep 18 09:59:55 PM CEST 2024 a2: 205520319495969441743068, b2: 360722932941420830399416
Wed Sep 18 09:59:55 PM CEST 2024 DISCRETE LOGARITHM: 92221507219705345685350
Wed Sep 18 10:08:40 PM CEST 2024
Wed Sep 18 10:08:40 PM CEST 2024 real 366m57.450s
Wed Sep 18 10:08:40 PM CEST 2024 user 433m16.898s
Wed Sep 18 10:08:40 PM CEST 2024 sys 1m54.288s

```

Rys. 5.1. Wynik działania programu

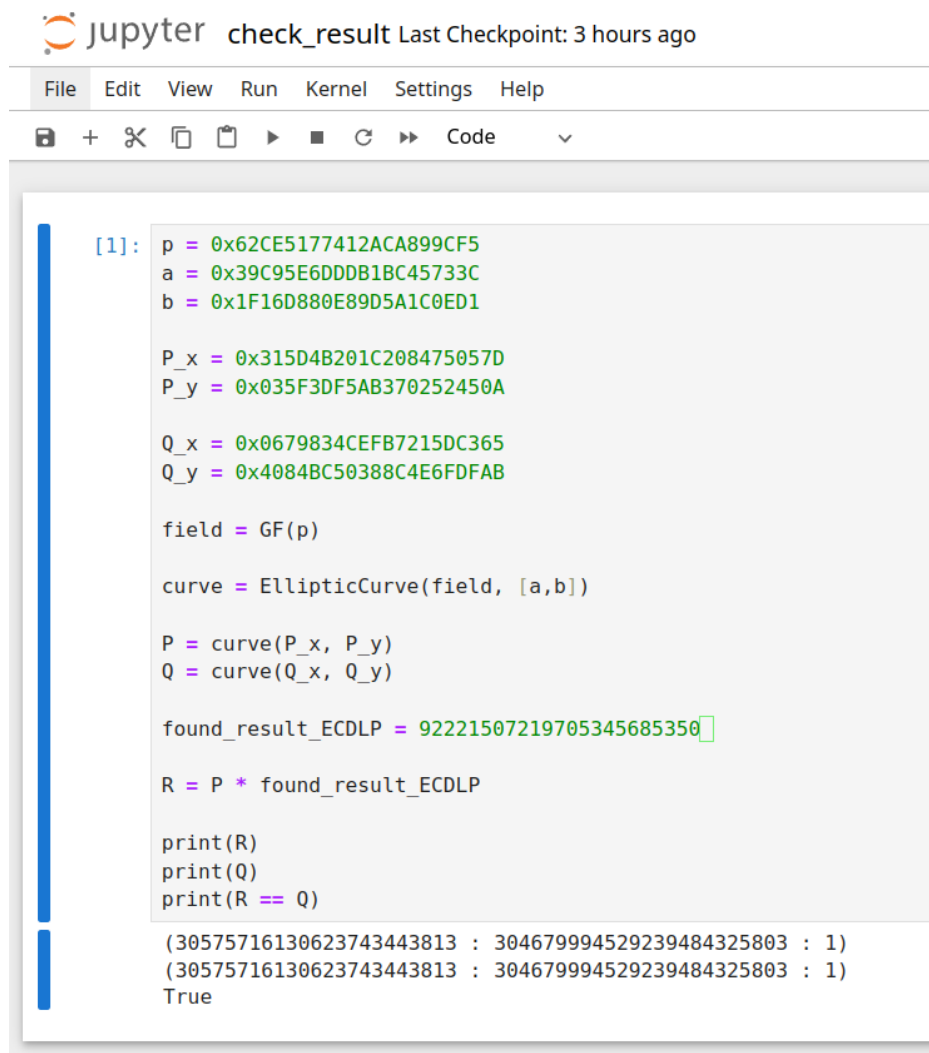
Poniżej są zaprezentowane wyniki pięciu niezależnych prób obliczenia ECDLP dla krzywej ECCp79, wskazujące liczbę znalezionych punktów wyróżnionych do momentu znalezienia kolizji oraz czas trwania obliczeń.

Tabela 5.2. Wyniki obliczeń ECDLP dla krzywej ECCp79

Próba	Liczba znalezionych punktów	Czas trwania obliczeń [s]
1	1856364	22195,42
2	791747	9898,74
3	770077	9329,74
4	1163047	13681,16
5	1207014	15001,94
Średnia	1153649	14021,00

5.3.1. Weryfikacja wyników za pomocą SageMath.

Aby upewnić się, że uzyskane rozwiązanie ECDLP jest poprawne, przeprowadzono weryfikację przy użyciu oprogramowania SageMath. Weryfikacja polegała na podstawieniu uzyskanego wyniku logarytmu dyskretnego do równania punktowego na krzywej eliptycznej i sprawdzeniu, czy generuje on punkt zgodny z założeniami wyzwania Certicom ECCp79.



```

jupyter check_result Last Checkpoint: 3 hours ago
File Edit View Run Kernel Settings Help

[1]: p = 0x62CE5177412ACA899CF5
      a = 0x39C95E6DDDB1BC45733C
      b = 0x1F16D880E89D5A1C0ED1

      P_x = 0x315D4B201C208475057D
      P_y = 0x035F3DF5AB370252450A

      Q_x = 0x0679834CEFB7215DC365
      Q_y = 0x4084BC50388C4E6FDFAB

      field = GF(p)

      curve = EllipticCurve(field, [a,b])

      P = curve(P_x, P_y)
      Q = curve(Q_x, Q_y)

      found_result_ECDLP = 92221507219705345685350

      R = P * found_result_ECDLP

      print(R)
      print(Q)
      print(R == Q)

(30575716130623743443813 : 304679994529239484325803 : 1)
(30575716130623743443813 : 304679994529239484325803 : 1)
True

```

Rys. 5.2. Weryfikacja wyniku w SageMath

Weryfikacja potwierdziła poprawność uzyskanego wyniku, co dowodzi skuteczności implementacji oraz poprawności algorytmu w zastosowaniu do krzywej ECCp79.

5.4. Podsumowanie wyników.

Na podstawie powyższych wyników można obliczyć średnią liczbę punktów wyróżnionych znajdujących przed wystąpieniem kolizji. Porównanie wyniku z oczekiwanymi wartościami teoretycznymi, pozwoliło potwierdzić osiągnięcie odpowiedniej losowości przy obliczaniu kolejnych kroków algorytmu. Oczekiwaną liczbę punktów do znalezienia kolizji można oszacować na podstawie wzoru:

$$E = \frac{\sqrt{\pi \cdot n/2}}{2^k}$$

gdzie n to liczność grupy punktów na krzywej eliptycznej, a k to liczba bitów zerowych które decydują czy dany punkt jest wyróżniony. W przypadku krzywej ECCp79, dla $n \approx 2^{79}$,

oraz punktów wyróżnionych których ostatnie 20 bitów to 0, oczekiwana liczba punktów potrzebna do znalezienia kolizji:

$$\frac{\sqrt{\pi \cdot 2^{79}/2}}{2^{20}} = 929262.5811$$

Uzyskany wynik na poziomie 1153649 jest bliski wartości oczekiwanej. Rozbieżność może wynikać z niewielkiej liczby przeprowadzonych prób oraz ograniczonej liczby punktów wstępnie obliczonych. Wydajność obliczeń jest zgodna z wynikami otrzymanymi podczas testów wydajności na ograniczonej liczbie iteracji algorytmu. Najszybsze znalezienie logarytmu dyskretnego zajęło mniej niż 3 godziny, co jest bardzo zadowalającym wynikiem.

5.5. Porównanie wyników

Praca najbardziej zbliżona tematycznie do niniejszej, to *Solving prime-field ECDLPs on GPUs with OpenCL* z 2015 roku, autorstwa Erika Boss'a [7]. Implementacja wykorzystana przez Boss'a opierała się na technologii OpenCL i była zoptymalizowana pod kątem kart graficznych AMD, jednak również dotyczyła problemu ECDLP dla krzywych modulo liczby pierwsze z Certicom Challenge. Praca była zoptymalizowana pod konkretną kartę graficzną oraz zastosowała technikę *Negation Map* która pozwala skrócić czas obliczeń o współczynnik $\sqrt{2}$ [3]. Osiągnęła ona wydajność 109 milionów iteracji dodawania punktów na sekundę dla karty AMD. Wyniki uzyskane przez Bossa wskazują, że obliczenie problemu logarytmu dyskretnego dla krzywej 112-bitowej zajęłoby odpowiednio 19,81 lat dla karty AMD HD7850 i 31,77 lat dla karty NVIDIA GTX 780.

Dla porównania, implementacja w niniejszej pracy przy użyciu karty NVIDIA GTX 2070 Super, uzyskała wydajność na poziomie 87,7 milionów iteracji na sekundę, więc jest to wydajność zbliżoną rzędem wielkości do wyniku AMD z pracy Boss'a. Ekstrapolując wydajność z działania na krzywej 79-bitowej, rozwiązanie ECDLP dla krzywej 112-bitowej zajęłoby około 32 lat na karcie NVIDIA GTX 2070 Super:

$$\frac{\sqrt{\pi \cdot \frac{2^{112}}{2}}}{87.7 \cdot 10^6 \cdot 3600 \cdot 24 \cdot 365} = 32.65$$

W przypadku kart graficznych Nvidia, oba wyniki są prawie identyczne, pomimo nowszej generacji karty graficznej użytej niniejszej pracy.

Wiele innych prac o podobnej tematyce również wykorzystuje GPU lub układy FPGA do obliczania logarytmu dyskretnego. Jednak zazwyczaj dotyczą one krzywej eliptycznej na ciele binarnym [33, 16, 21]. W takim przypadku porównanie traci na wartości, ponieważ na ciele binarnym można zastosować więcej optymalizacji przy operacjach na krzywej [5].

W celach historycznych warto wspomnieć, że pierwsze złamanie krzywej ECCp79 udało się w 1997 roku [9]. W tym celu wykorzystano kilka komputerów współpracujących ze sobą, z czego pojedyncza stacja osiągała wydajność na poziomie 25 bilionów operacji

dodawania punktów na krzywej w ciągu dnia. Daje to około 289351,85 operacji na sekundę, w porównaniu do 87,7 milionów operacji na sekundę w tej implementacji.

6. Podsumowanie oraz dalsze usprawnienia

Celem tej pracy było zaimplementowanie systemu, który z wykorzystaniem GPU jest w stanie rozwiązać ECDLP na ECCp79 w jak najkrótszym czasie.

Aby osiągnąć ten cel, niezbędne było zapoznanie się z wieloma publikacjami, które przedstawiają współczesne sposoby optymalizacji obliczeń kryptograficznych oraz najwydajniejsze techniki projektowania systemów do kryptoanalizy krzywych eliptycznych. Pozwoliło to na zastosowanie bardziej optymalnego algorytmu Rho Pollard'a, w wersji Addition walk, który znacznie lepiej działa w przypadku architektury SIMD jaką jest GPU. Wykorzystanie optymalizacji, takich jak Montgomery Trick, również pozwoliło jeszcze bardziej zwiększyć wydajność obliczeń.

Opracowana koncepcja oraz opis implementacji, szczegółowo przedstawiają sposób działania zarówno głównego programu do obliczeń kryptograficznych, jak i serwera wraz z całą architekturą niezbędną do uruchomienia całego systemu. Wszelkie napotkane trudności oraz decyzje projektowe, zostały dokładnie opisane, co pozwala na odtworzenie podobnego systemu oraz wyjaśnia niektóre zabiegi implementacji. Wszystkie moduły składające się na system, zostały dokładnie przetestowane oraz zweryfikowane przy pomocy szerokiej gamy testów. Dodatkowo działanie zostało zweryfikowane poprzez prawidłowe rozwiązanie założonego problemu ECDLP.

Praca spełniła wszystkie założone cele, osiągając przy tym satysfakcjonujące wyniki. Poprawne rozwiązanie wyzwania Certicom, przy jednoczesnym osiągnięciu wydajności porównywalnej z podobnymi pracami, stanowi istotny sukces. Realizacja pracy pozwoliła na zgłębienie zagadnień związanych ze współczesną kryptografią, jej praktyczną implementacją oraz technikami akceleracji obliczeń z wykorzystaniem GPU.

Wyniki zademonstrowane w pracy pokazały, że GPU umożliwia osiągnięcie wysokiej wydajności w obliczeniach kryptograficznych. Pomimo rosnącej mocy obliczeniowej, rozmiar kluczy stosowanych we współczesnej kryptografii nadal zapewnia odpowiedni poziom bezpieczeństwa. W efekcie złamanie kryptosystemów opartych na ECC w rozsądnym czasie, nawet wykorzystując wiele układów GPU, pozostaje praktycznie niemożliwe.

6.1. Krytyczna ocena oraz dalsze usprawnienia

Pomimo uzyskania zadowalających wyników, w pracy nadal pozostaje przestrzeń na dalsze optymalizacje i usprawnienia.

Zastosowanie lżejszych bibliotek niż pakiet SageMath mogłoby uprościć uruchamianie rozwiązania na innych platformach, eliminując konieczność czasochłonnej kompilacji rozbudowanego pakietu obliczeniowego, którego znaczna część pozostaje niewykorzystana.

Wprowadzenie redukcji Barrett'a do operacji dzielenia modulo p w programie GPU, mogłoby dodatkowo zwiększyć wydajność obliczeń w ciele. Ponieważ wszystkie opera-

cje wykonywane są na tych samych parametrach ciała, przeliczenie specjalnej wartości wymaganej przez tę technikę byłoby konieczne jedynie na początku obliczeń, znacznie upraszczając implementację. Dodatkowo, odpowiednie zastosowanie Negation Map, pozwoliłoby znacząco skrócić czas obliczeń. Jednak zaimplementowanie tej techniki, nie jest trywialne w architekturze SIMD [3].

Kolejnym miejscem na potencjalne usprawnienia jest optymalizacja wykorzystania rejestrów w kernel'u CUDA. Obecna implementacja ma niewielką liczbę rejestrów, które musiały być przeniesione przez kompilator do wolniejszej pamięci globalnej. Prawdopodobnie uważana optymalizacja oraz dokładniejsza analiza z pomocą profiler'a *NVIDIA Nsight Systems* pozwoliłaby zredukować ilość *spilled registers*, ograniczając narzut związany z niższą przepustowością pamięci globalnej, oraz zmniejszyć liczbę rozgałęzień w kodzie.

Mniej istotnym z punktu widzenia wydajności (obliczeniowej), ale możliwym do poprawy miejscem, jest organizacja kodu. Z racji *jednorazowego* charakteru tej implementacji, czystość i struktura kodu nie była priorytetem podczas jej rozwoju. Osoba bardziej doświadczona w środowisku programowania C++ oraz CUDA, prawdopodobnie znalazłaby kilka miejsc, w których zastosowanie innego podejścia dałoby lepsze rezultaty oraz ułatwiłoby potencjalną modyfikację.

Bibliografia

- [1] Sio-Long. Ao, Len. Gelman i David W. L. Hukins. *Security Analysis of Elliptic Curve Cryptography and RSA*. Newswood Limited, 2016, s. 1210. ISBN: 9789881925305.
- [2] Elaine Barker. *Recommendation for Key Management Part 1: General*. Sty. 2016. DOI: 10.6028/NIST.SP.800-57pt1r4. URL: <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r4.pdf>.
- [3] Daniel J. Bernstein, Tanja Lange i Peter Schwabe. *On the correct use of the negation map in the Pollard rho method*. Cryptology ePrint Archive, Paper 2011/003. 2011. URL: <https://eprint.iacr.org/2011/003>.
- [4] Daniel J Bernstein i in. „ECC2K-130 on NVIDIA GPUs”. W: (2012). URL: <http://www.ecc-challenge.info>.
- [5] Ian F Blake, Gadiel Seroussi i Nigel Paul Smart. *Krzywe eliptyczne w kryptografii*. 2005. ISBN: 9788320429510.
- [6] Joppe W. Bos i in. „ECC2K-130 on Cell CPUs”. W: 2010, s. 225–242. DOI: 10.1007/978-3-642-12678-9_14.
- [7] Erik Boss. *Solving prime-field ECDLPs on GPUs with OpenCL*. 2015.
- [8] *Certicom Challenge*. <https://www.certicom.com/content/certicom/en/the-certicom-ecc-challenge.html>. Accessed: 2024-08-20.
- [9] *Certicom Cracked*. <https://tbtf.com/resource/certicom6.html>. Accessed: 2024-12-30.
- [10] Kaushal A Chavan, Indivar Gupta i Dinesh B Kulkarni. „A Review on Solving ECDLP over Large Finite Field Using Parallel Pollard’s Rho (p) Method”. W: 18 (2), s. 1–11. DOI: 10.9790/0661-1802040111. URL: www.iosrjournals.org.
- [11] John Cheng, Max Grossman i Ty McKercher. *Professional CUDA C Programming*. Wrox Press, 2014, s. 528. ISBN: 9781118739273.
- [12] Andrzej Chrzęszczyk. *Algorytmy teorii liczb i kryptografii w przykładach*. 2010, s. 328. ISBN: 9788360233672.
- [13] *CUDA C++ Programming Guide*. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Accessed: 2024-09-30.
- [14] Gueric Meurice De Dormale, Philippe Bulens i Jean Jacques Quisquater. „Collision search for elliptic curve discrete logarithm over GF(2 m) with FPGA”. W: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 4727 LNCS (2007), s. 378–393. ISSN: 03029743. DOI: 10.1007/978-3-540-74735-2_26.
- [15] Junfeng Fan i in. „Breaking elliptic curve cryptosystems using reconfigurable hardware”. W: *Proceedings - 2010 International Conference on Field Programmable Logic and Applications, FPL 2010* (2010), s. 133–138. DOI: 10.1109/FPL.2010.34.
- [16] Tim Güneysu i Christof Paar. „Ultra high performance ECC over NIST primes on commercial FPGAs”. W: *Lecture Notes in Computer Science (including subseries*

- Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics*) 5154 LNCS (2008), s. 62–78. ISSN: 03029743. DOI: 10.1007/978-3-540-85053-3_5.
- [17] Badis Hammi i in. „A Lightweight ECC-Based Authentication Scheme for Internet of Things (IoT)”. W: *IEEE Systems Journal* 14 (3 wrz. 2020), s. 3440–3450. ISSN: 19379234. DOI: 10.1109/JSYST.2020.2970167.
 - [18] Ryan Henry i Ian Goldberg. „Solving Discrete Logarithms in Smooth-Order Groups with CUDA 1”. W: (). URL: <http://cacr.uwaterloo.ca/>.
 - [19] Sanders Jason i Kandrot Edward. *Cuda by example*. Addison-Wesley Professional, 2011, s. 313. ISBN: 9786612660122.
 - [20] Lyndon Judge, Suvarna Mane i Patrick Schaumont. „A hardware-accelerated ECDLP with high-performance modular multiplication”. W: *International Journal of Reconfigurable Computing* 2012 (2012). ISSN: 16877195. DOI: 10.1155/2012/439021.
 - [21] Piotr Majkowski i in. „Heterogenic distributed system for cryptanalysis of elliptic curve based cryptosystems”. W: *Proceedings of 19th International Conference on Systems Engineering, ICSEng 2008* (2008), s. 300–305. DOI: 10.1109/ICSENG.2008.73.
 - [22] Suvarna Mane, Lyndon Judge i Patrick Schaumont. „An Integrated Prime-Field ECDLP Hardware Accelerator with High-Performance Modular Arithmetic Units”. W: *2011 International Conference on Reconfigurable Computing and FPGAs*. IEEE, list. 2011, s. 198–203. ISBN: 978-0-7695-4551-6. DOI: 10.1109/ReConFig.2011.12. URL: <http://ieeexplore.ieee.org/document/6128577/>.
 - [23] Alfred J. Menezes, Paul C. Van Oorshot i Scott A Vanstone. „Handbook of Applied Cryptography”. W: (2001).
 - [24] Peter L Montgomery. „Speeding the Pollard and elliptic curve methods of factorization”. W: *Mathematics of Computation* 48 (1987), s. 243–264. URL: <https://api.semanticscholar.org/CorpusID:4262792>.
 - [25] Paul C Van Oorschot i Michael J Wiener. *Parallel Collision Search with Cryptanalytic Applications*. 1999.
 - [26] Jairo Panetta i in. „Scalability of CPU and GPU Solutions of the Prime Elliptic Curve Discrete Logarithm Problem”. W: *2017 29th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*. IEEE, paź. 2017, s. 33–40. ISBN: 978-1-5090-1233-6. DOI: 10.1109/SBAC-PAD.2017.12.
 - [27] J M Pollard. „Monte Carlo Methods for Index Computation (mod p)”. W: 32 (143 1978), s. 918–924.
 - [28] *Python Wiki*. <https://wiki.python.org/moin/GlobalInterpreterLock>. Accessed: 2024-08-20.
 - [29] Shreyas Srinath, G S Nagaraja i Ramesh Shahabadkar. „A detailed Analysis of Lightweight Cryptographic techniques on Internet-of-Things”. W: *2021 IEEE International Conference on Computation System and Information Technology for Sustainable Solutions (CSITSS)*. IEEE, grud. 2021, s. 1–6. ISBN: 978-1-6654-0610-9. DOI: 10.1109/CSITSS54238.2021.9683091.

- [30] Douglas R. Stinson i Maura B. Paterson. *Kryptografia. W teorii i praktyce*. IV. 2021.
- [31] Edlyn Teske. „ON RANDOM WALKS FOR POLLARD’S RHO METHOD”. W: *MATHEMATICS OF COMPUTATION* 70 (234 2000).
- [32] Vidyotma Thakur i in. „Cryptographically secure privacy-preserving authenticated key agreement protocol for an IoT network: A step towards critical infrastructure protection”. W: *Peer-to-Peer Networking and Applications* 15 (1 sty. 2022), s. 206–220. ISSN: 19366450. DOI: 10.1007/s12083-021-01236-w.
- [33] Erich Wenger i Paul Wolfger. „Solving the discrete logarithm of a 113-bit Koblitz curve with an FPGA cluster”. W: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8781 (2014), s. 363–379. ISSN: 16113349. DOI: 10.1007/978-3-319-13051-4_22/TABLES/5. URL: https://link.springer.com/chapter/10.1007/978-3-319-13051-4_22.

Wykaz symboli i skrótów

EiTI – Wydział Elektroniki i Technik Informatycznych

PW – Politechnika Warszawska

FPGA – Field Programmable Gate Array

CUDA – Compute Unified Device Architecture

ECC – Elliptic Curve Cryptography

DLP – Discrete Logarithm Problem

GF – Galois Field (ciało skończone)

Spis rysunków

2.1. Krzywa eliptyczna $y^2 = x^3 - 4x + 2$ nad ciałem liczb rzeczywistych	11
2.2. $P + Q$ na krzywej eliptycznej $y^2 + y = x^3 - x^2 + 2x$	12
2.3. Krzywa eliptyczna $y^2 = x^3 - 4x + 2$ nad $GF(2^{11} - 9)$	13
3.1. Schemat architektury	19
4.1. Schemat działania jednego kroku pętli z wykorzystaniem okna	31
4.2. Tail effect. Wątki kończą pracę po znalezieniu jednego punktu wyróżnionego.	32
4.3. Tail effect. Wątki kończą pracę po znalezieniu 3 punktów wyróżnionych.	32
4.4. Suma znalezionych punktów wyróżnionych z podziałem na strumienie.	33
5.1. Wynik działania programu	39
5.2. Weryfikacja wyniku w SageMath	41

Spis tabel

5.1. Wyniki testów wydajności	39
5.2. Wyniki obliczeń ECDLP dla krzywej ECCp79	40

Spis wydruków

4.1 Prototyp funkcji wejściowej programu CUDA	22
4.2 Struktura do przechowywania długich liczb	23
4.3 Mniejszy typ danych do długich liczb	23
4.4 Odejmowanie modulo	24
4.5 Prototypy funkcji wykorzystywanych do operacji na długich liczbach	26
4.6 Inicjalizacja pamięci współdzielonej i flagi <code>warp_finished</code>	27

4.7	Funkcja przydzielająca punkt	28
4.8	Generacja punktów startowych	35
4.9	Obliczanie współczynników a i b	36

Spis załączników