

Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH



Instytut Cyberbezpieczeństwa

Praca dyplomowa inżynierska

na kierunku Telekomunikacja
w specjalności Techniki Teleinformatyczne

Akceleracja sprzętowa kryptoanalizy algorytmów kryptograficznych

Andrzej Tłomak

Numer albumu 311450

promotor

dr. hab. inż. Mariusz Rawski

WARSZAWA 2025

Akceleracja sprzętowa kryptoanalizy algorytmów kryptograficznych

Streszczenie. Cel pracy

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

Hardware acceleration of cryptanalysis of cryptographic algorithms

Abstract. The objective of this phase included a review of the literature describing the current State-of-Art in cryptanalysis of systems based on Elliptic curves in Finite Fields. It involved getting deeper knowledge of theory and mathematical foundation of Elliptic curves as well as setting up an environment to develop implementation utilizing CUDA technology.

Keywords: Elliptic curves, Cryptography, Cryptanalysis, CUDA, FPGA, rho Pollard algorithm



.....
miejscowość i data

.....
imię i nazwisko studenta
.....
numer albumu
.....
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanej z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

.....
czytelny podpis studenta

Spis treści

1. Wprowadzenie	9
1.1. Cel pracy	9
2. Wstęp teoretyczny	10
2.1. Procesor graficzny	10
2.1.1. Model programowania CUDA	10
2.1.2. Hierarchia pamięci	11
2.2. Krzywe eliptyczne	11
2.2.1. Dodawanie punktów na krzywej eliptycznej	12
2.2.2. Krzywe eliptyczne na ciele skończonym	13
2.2.3. Problem logarytmu dyskretnego	14
2.2.4. Problem logarytmu dyskretnego na krzywej eliptycznej	15
2.3. Algorytm Rho Pollard’a	15
2.3.1. Równoległy algorytm Rho Pollarda	16
2.3.2. Adding Walk	16
3. Koncepcja	18
3.1. Klient	18
3.2. Serwer	19
3.3. Przebieg działania	19
4. Implementacja	21
4.1. Narzędzia oraz sprzęt	21
4.1.1. Sprzęt	21
4.1.2. Program po stronie CPU	21
4.1.3. System budowania oraz kompilacja CUDA C++	21
4.1.4. Testy	21
4.2. Program klienta Python	22
4.3. Program klienta CUDA	22
4.4. Artytmetyka na ciele F_p	23
4.4.1. Reprezentacja długich liczb	23
4.4.2. Operacje na długich liczbach	24
4.4.3. Dodawanie i odejmowanie modulo	24
4.4.4. Mnożenie modulo	25
4.4.5. Odwrotność modulo	25
4.4.6. Biblioteka CGBN	27
4.5. Funkcja iterująca	28
4.5.1. Wstępnie obliczone punkty	28
4.5.2. Punkty wyróżnione	29
4.5.3. Obliczanie odwrotności w seriach	30

4.6. Tail effect	32
4.7. Program serwera	35
Generacja punktów startowych	36
4.7.1. Kolizje	37
4.8. Logowanie	37
5. Wyniki	38
5.1. Testy implementacji	38
5.2. Wydajność	39
5.3. Rozwiązanie ECDLP dla krzywej ECCp79	40
6. Porównanie wyników	42
Bibliografia	43
Wykaz symboli i skrótów	46
Spis wydruków	47
Spis załączników	47

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)* 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, takich jak urządzenia IoT czy urządzenia mobilne [29, 15].

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, dlatego kryptoanaliza ECC wymaga dużych zasobów obliczeniowych. Z tego powodu, często wykorzystywane w tej roli są układy GPU [7, 24, 4], FPGA [30, 20, 14, 19] czy nawet konsole dla graczy [6].

Gwałtowny rozwój kart graficznych (GPU) 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 Pollarda, 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 pod obliczenie logarytmu dyskretnego na krzywej eliptycznej ECCp-79, zaproponowanej w challenge'u Certicom.

Głównymi elementami stworzonego systemu jest program klienta wykonującego część algorytmu Rho Pollarda 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 GPGPU (*general purpose GPU*) w dziedzinach takich jak obliczenia naukowe, machine learning czy kryptografia.

W mojej pracy, w celu przeprowadzenia ataku na krzywą eliptyczną, wykorzystałem algorytm *Rho Pollard'a*, który z niewielkimi modyfikacjami można bardzo skutecznie wykonywać równolegle. Z tego powodu wykorzystanie GPU do kryptoanalizy, pozwala znacznie przyspieszyć czas obliczeń. W celu stworzenia programu na GPU wykorzystałem technologię 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'a* - 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* [11]. Następnie bloki są uruchamiane na dostępnych *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.

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 [11]. 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

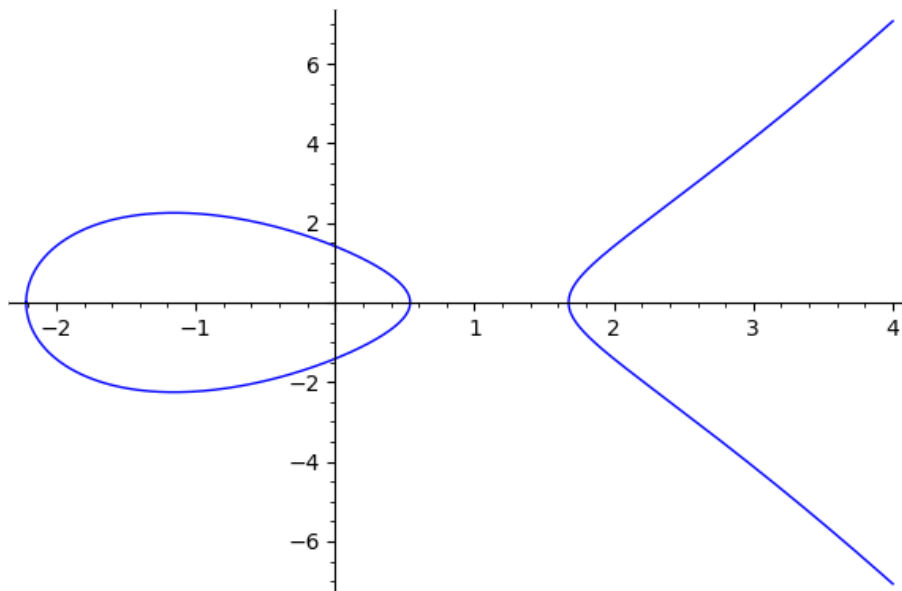
Zakładając, że ciało \mathbb{K} ma charakterystykę różną od 2 i 3, oraz że stałe $a, b \in \mathbb{K}$ spełniają warunek:

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

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 [27].



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

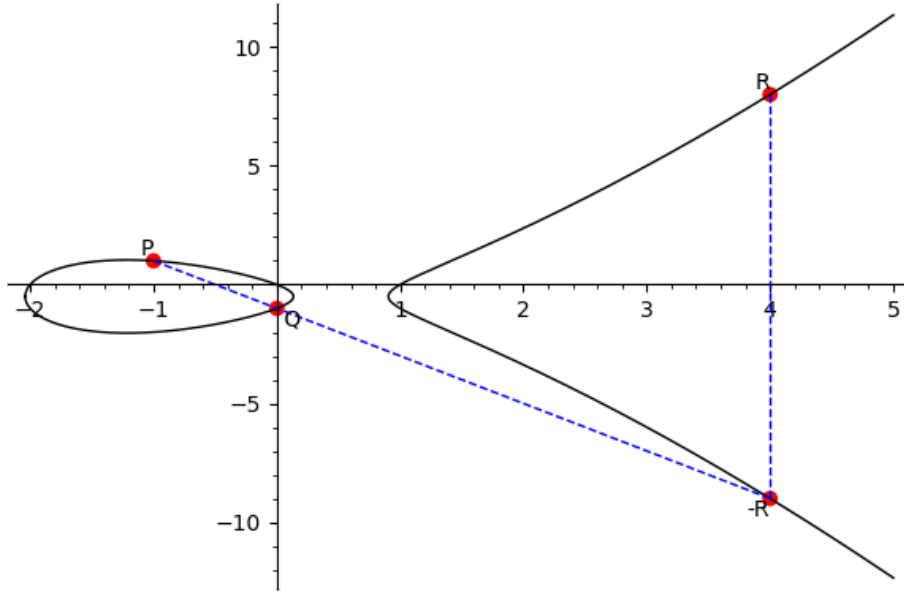
Krzywe eliptyczne zdefiniowane na liczbach rzeczywistych nie są kluczowe w systemach kryptograficznych [27], ale takie ustawienia pozwalają na prostsze przedstawienie niektórych zagadnień np. dodawanie punktów na krzywej.

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 . 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 $2P$ [10][27].

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



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

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

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

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

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

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 $P = -Q$:

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

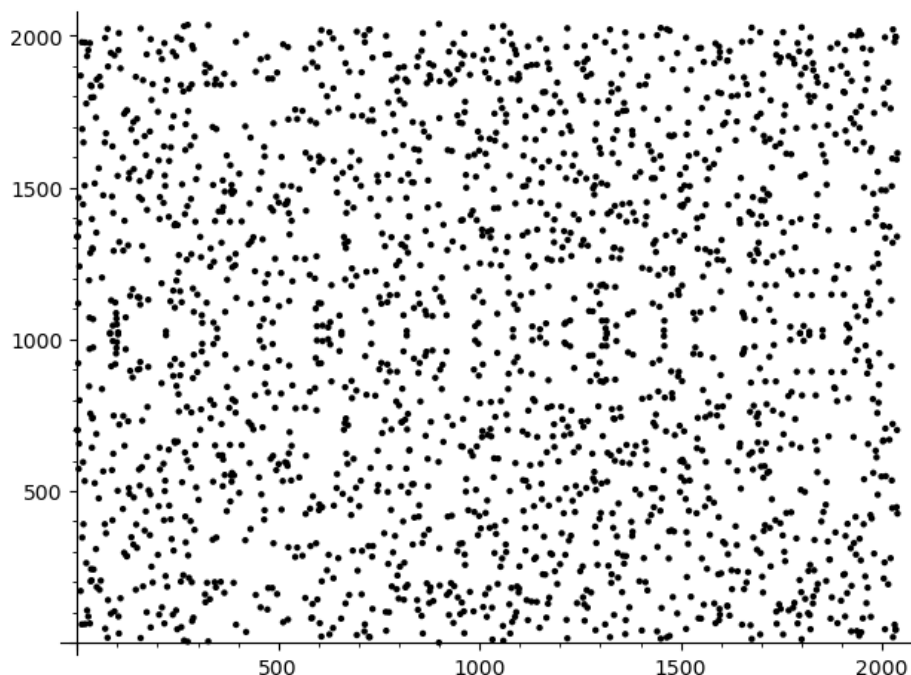
Dodatkowo odwrotność punktu na krzywej P definiujemy jako $-P = (x, -y)$ [27].

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. W swojej pracy skupiłem się wyłącznie na krzywych zdefiniowanych na ciele skończonym F_p gdzie p jest liczbą pierwszą.

Wykres krzywej eliptycznej nad ciałem F_p 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 $y^2 = x^3 - 4x + 2$ nad $GF(2^{11} - 9)$

2.2.3. Problem logarytmu dyskretnego

Problem logarytmu dyskretnego (**DLP**) jest podstawą kryptosystemów oparych o grupy. Jednymi z bardziej znanych są kryptosystem ElGamala oraz protokół wymiany kluczy Diffie-Hellmana'a.

Problem logarytmu dyskretnego można zdefiniować na grupach cyklicznych. zarówno na grupie multiplikatywnej (\mathbb{G}, \cdot) oraz grupie addytywnej $(\mathbb{G}, +)$, przy odpowiednim zdefiniowaniu działań grupowych.

Jeżeli G jest 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, ponieważ nie istnieje algorytm, który znajduje x w czasie wielomianowym[27].

2.2.4. Problem logarytmu dyskretnego na krzywej eliptycznej

W przypadku kryptografii opartej o krzywe eliptyczne, DLP dotyczy cykliczej 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 multiplikacyjnych, problem logarytmu dyskretnego na krzywej eliptycznej polega na znalezieniu x .

Przy odpowiednim wyborze grupy addytywnej, rozwiązanie problemu logarytmu dyskretnego, tj. znalezienie x , jest trudne [27].

2.3. Algorytm Rho Pollard'a

Zaproponowany przez Johna Pollard'a w 1978 roku [25], 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 [27], algorytm Rho Pollard'a jest bardziej efektywny pamięciowo, nie wymagając przestrzeni $O(\sqrt{n})$ a jedynie $O(1)$ w wersji sekwencyjnej [27][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, iteracyjne 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 $R = aP + bQ$:

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

$S_1 \cup S_2 \cup S_3$ 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, często stosuje się algorytm poszukiwania cyklu Floyd'a. W takim przypadku, obliczamy w każdej iteracji obliczamy 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 [23]. Dlatego w swojej pracy wykorzystałem równoległą wersję algorytmu, zaproponowaną przez Van Oorschota i Wienera [23].

2.3.1. Równoległy algorytm Rho Pollarda

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 Pollard's Rho do obliczania kolejnych punktów na krzywej eliptycznej. 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 [28], 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 definiujemy 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 [28].

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 (Single Instruction, Multiple Data). 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 wyróżnialnych 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.

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 zwielokrotnić 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 w 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 działa on aż do znalezienia rozwiązania i zakończenia wszystkich wątków przez serwer. W trakcie działania, 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 działaniem 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 kluczowe do późniejszego obliczenia logarytmu dyskretnego. Mechanizm ten pozwala na precyzyjne wyliczenie logarytmu dyskretnego po wykryciu kolizji, dzięki możliwości odwzorowania znalezionych punktów na kombinację wielokrotności tych punktów bazowych.

3.3. Przebieg działania

W momencie uruchomienia programu, serwer generuje wyznaczoną ilość 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

1. Generacja nowych punktów startowych
2. Przekazanie punktów startowych do kolejki zadań
3. Pobranie znalezionych punktów wyróżnionych z kolejki z wynikami

3. Koncepcja

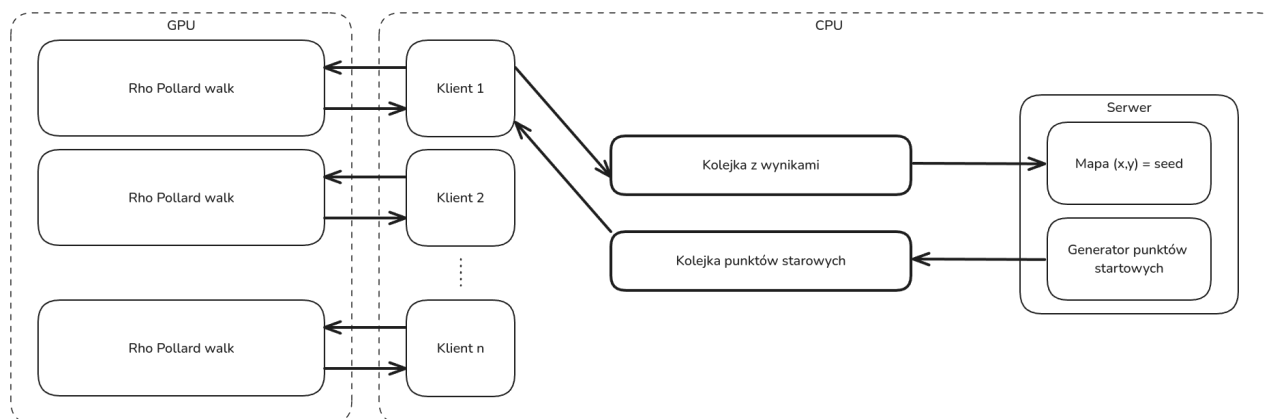
4. Sprawdzenie czy w bazie punktów znalezionych wystąpiła kolizja
5. Dodanie punktów do bazy punktów znalezionych

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.

Większość opisanych zagadnień dotyczy części klienta napisanej w CUDA C++, poza sekcją na końcu, poświęconą serwerowi oraz części klienta działającej na CPU.

4.1. Narzędzia oraz sprzęt

4.1.1. Sprzęt

- 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++, wykorzystałem 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 Pollarda. 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*:

```
def GPUworker(  
    starting_params: StartingParameters ,  
    task_queue: Queue,  
    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, 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:

```
LIMBS = 5  
class bn(ctypes.Structure):  
    _fields_ = [("array", ctypes.c_uint32 * LIMBS)]  
  
class small_bn(ctypes.Structure):  
    _fields_ = [("array", ctypes.c_uint32 * ceil(LIMBS / 2))]  
class EC_point(ctypes.Structure):  
    _fields_ = [  
        ("x", bn),  
        ("y", bn),  
        ("seed", bn),  
        ("is_distinguish", ctypes.c_uint32),  
    ]
```

4.3. Program klienta CUDA

Program uruchamiany na GPU został skompilowany jako dynamicznie linkowana biblioteka (*SHARED*). Aby zapewnić zgodność z interfejsem binarnym (ABI) wykorzystywa-

nym przez bibliotekę Ctypes, do głównej funkcji dodano słowo kluczowe `extern "C"`. 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. Prototyp funkcji wejściowej do programu CUDA:

```
extern "C" {
void run_rho_pollard (
    EC_point *startingPts ,
    uint32_t instances ,
    uint32_t n, PCMP_point *precomputed_points ,
    EC_parameters *parameters ,
    int stream
)
```

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

4.4. Artrytmetyka na ciele F_p

4.4.1. Reprezentacja długich liczb

W przypadku ciała F_p , gdzie p jest liczbą pierwszą o rozmiarze 79 bit, potrzebujemy co najmniej 79 bitowego typu danych, do samego przechowywania liczb. Jednak nawet 79 bitowy typ danych nie wystarczy, jeżeli chcemy 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, musiałem wykorzystać 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_t`:

```
struct bn
```

```
{  
    uint32_t array[5];  
};
```

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, wykorzystuję następujący typ danych:

```
struct small_bn  
{  
    uint32_t array[3];  
};
```

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, odpowiednio dostosowałem małą bibliotekę dostępną 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 [21] wykorzystywane przy obliczeniach *multiple precision*. Dużą zaletą tej biblioteki jest jej prostota i brak wykorzystywania 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, zaimplementowałem 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 odejmowanie 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 prowadzimy obliczenia 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. Na szczęście, możemy bardzo łatwo wrócić do poprawnego wyniku wykorzystując jedną operację dodawania. Korzystając z faktu, że wszystkie podstawowe operacje są wykonywane modulo $2^{32 \cdot n}$:

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

Przykładowa implementacja z wykorzystaniem *tiny-bignum-c*:

```
bignum_sub(a, b, c);
if (bignum_cmp(a, b) == SMALLER)
{
    bignum_add(c, p, temp);
    bignum_assign(c, temp);
}
```

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 swojej pracy przeprowadzam standardowe mnożenie modularne, jednak również istnieją bardziej wydajne sposoby, na przykład wykorzystanie redukcji Barreta CITE APPLIED.

Aby przeprowadzić klasyczne mnożenie modularne ciele F_p , na początku musimy przeprowadzić standardowe mnożenie długich liczb. Następnie, otrzymany wynik dzielimy przez p i zwracamy resztę 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 zaimplementowałem 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.

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
```

4.4.6. Biblioteka CGBN

W pierwotnej wersji swojej pracy, do operacji na dużych liczbach wykorzystałem specjalną bibliotekę CGBN dla platformy CUDA. Oferuje ona wszystkie podstawowe operacje na długich liczbach, takie jak dodawania, odejmowanie czy mnożenie nawet do 32 tys. bitów. Dodatkowo, posiada ona implementację bardziej zaawansowanych funkcji, takich jak odwrotność modulo czy redukcja Barreta. Niestety, biblioteka ta narzuca spore ograniczenia pod kątem zasobów. Niezbędne jest grupowanie wątków w grupy 4, 8, 16 lub 32. Wysoką wydajność obliczeń, uzyskiwałem 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*.

Wydruk 4.1. Prototypy funkcji wykorzystywanych do operacji na długich liczbach

```
__device__ void bignum_init(struct bn *n);
__device__ void bignum_from_int(struct bn *n, DTYPE_TMP i);
__device__ int bignum_to_int(struct bn *n);

__device__ void bignum_add(struct bn *a, struct bn *b, struct bn *c);
__device__ void bignum_sub(struct bn *a, struct bn *b, struct bn *c);
__device__ void bignum_mul(struct bn *a, struct bn *b, struct bn *c);
__device__ void bignum_div(struct bn *a, struct bn *b, struct bn *c);
__device__ void bignum_mod(struct bn *a, struct bn *b, struct bn *c);
__device__ void bignum_divmod(
    struct bn *a, struct bn *b, struct bn *c, struct bn *d
);

__device__ void bignum_assign_fsmall(struct bn *dst, struct small_bn *src);
__device__ void bignum_assign_small(struct small_bn *dst, struct small_bn *src);

__device__ void bignum_modinv(struct bn *a, struct bn *b, struct bn *c);

__device__ void bignum_and(struct bn *a, struct bn *b, struct bn *c);
__device__ void bignum_or(struct bn *a, struct bn *b, struct bn *c);
__device__ void bignum_xor(struct bn *a, struct bn *b, struct bn *c);
__device__ void bignum_lshift(struct bn *a, struct bn *b, int nbits);
__device__ void bignum_rshift(struct bn *a, struct bn *b, int nbits);
```

```
__device__ int bignum_cmp(struct bn *a, struct bn *b);
__device__ int bignum_is_zero(struct bn *n);
__device__ void bignum_inc(struct bn *n);
__device__ void bignum_dec(struct bn *n);
__device__ void bignum_assign(struct bn *dst, struct bn *src);
```

4.5. Funkcja iterująca

Główna funkcja realizująca kolejne kroki algorytmu Rho Pollarda została zaimplementowana jako kernel GPU. Na początku swojego działania kernel ładuje punkty wstępnie obliczone do pamięci współdzielonej (*shared memory*), która pełni funkcję pamięci podręcznej.

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*).

Przykładowy fragment kodu ilustrujący inicjalizację pamięci współdzielonej oraz obsługę flagi:

Wydruk 4.2. Inicjalizacja pamięci współdzielonej i flagi `warp_finished`

```
if (threadIdx.x == 0)
{
    printf("STREAM_%d_BLOCK_%d_started\n", stream, blockIdx.x);
    for (int i = 0; i < PRECOMPUTED_POINTS; i++)
    {
        bignum_assign_small(&SMEMprecomputed[i].x, &args.precomputed[i].x);
        bignum_assign_small(&SMEMprecomputed[i].y, &args.precomputed[i].y);
    }
    warp_finished = 0;
}
__syncthreads();
```

4.5.1. Wstępnie obliczone punkty

Dostęp do wstępnie obliczonych punktów jest niezbędny w każdej iteracji *Addition walk*., dlatego ważne jest, aby je przechowywać w szybkiej pamięci. Wykorzystałem w tym celu pamięć współdzieloną *shared memory*. W przeciwieństwie do znacznie większej pamięci globalnej, jest ona przechowywana bezpośrednio na SM [9, 17]. Ponieważ ilość punktów która zapewnia dostatecznie losowy spacer po krzywej eliptycznej jest

stosunkowo niewielka [28], to rozmiar pamięci nie stanowi problemu. 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.

Wydruk 4.3. Funkcja przydzielająca punkt

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

4.5.2. Punkty wyróżnione

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

Do sprawdzenia, czy punkt jest wyróżnionym, służy prosta funkcja obliczająca bitową operację AND ze współrzędnej punktu oraz specjalnej maski bitowej wyznaczanej 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.

4.5.3. Obliczanie odwrotności w seriach

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

Koszt dodawania punktów na krzywej:

$$1O + 3M$$

Najdroższym działaniem, które wpływa na wysoki koszt obliczeniowy mnożenia modularnego oraz odwrotności w ciele, jest operacja dzielenia [21]. Przykładowo, wykorzystywana przeze mnie 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 wewnątrz algorytmu 1. To czyni ją najdroższą operacją na ciele, znacznie kosztowniejszą niż operacja mnożenia modularnego.

Aby przyspieszyć obliczenia, zastosowałem technikę obliczania wielu odwrotności za jednym razem, znaną jako *Montgomery Trick* [22].

Idea stojąca za tym sposobem, jest następująca. Niech x_1, \dots, x_n będą elementami, których odwrotność chcemy policzyć. Na początku obliczamy tablicę 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, obliczamy 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 obliczamy 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, 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 zastosowałem 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 zaczynamy tracić zyski z metody Montgomeryego, a nawet zaczynamy działać wolniej, z powodu dodatkowych obliczeń, których nie wykorzystujemy. Wynika to z faktu, że po znalezieniu i punktów, zysk z metody jest postaci:

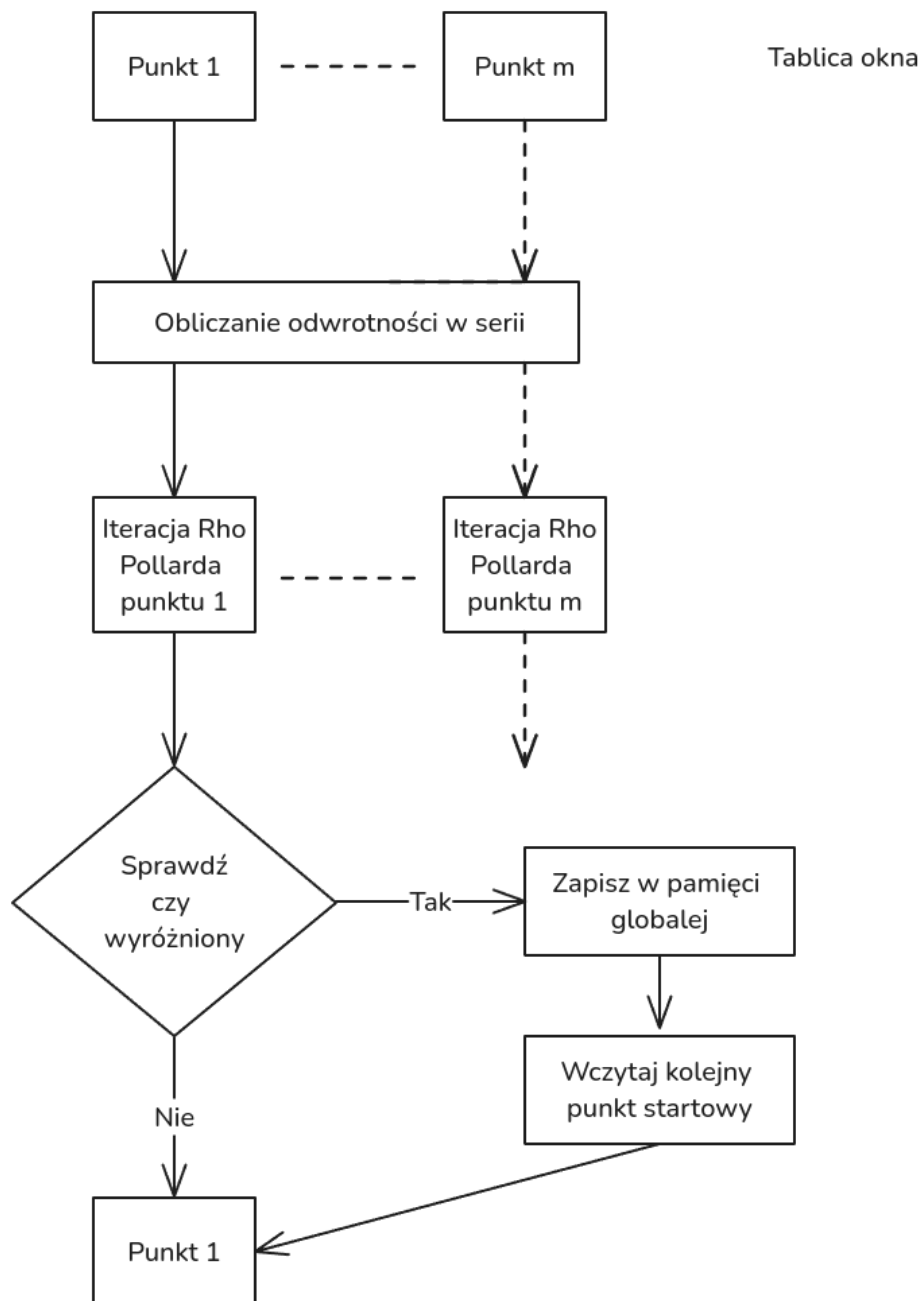
$$(n - i)O = O + 3(n - i)M + 3(i)M$$

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, zastosowałem okienkowanie obliczeń w tablicy o stałym rozmiarze m . Sposób ten wymaga, aby każda 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, ładujemy do tablicy okna kolejne m punktów startowych z pamięci globalnej. Następnie w pętli obliczamy odwrotności wymagane dla operacji dodawania punktów i przeprowadzamy dodawanie. Tym sposobem, w jednym kroku pętli, wykonujemy jedną iterację algorytmu Rho Pollarda dla m punktów. Na samym końcu każdego kroku pętli, sprawdzamy czy któryś z obliczonych punktów jest punktem wyróżnionym. Jeżeli tak, znaleziony punkt zapisujemy na pierwszym wolnym miejscu w pamięci globalnej, a na jego miejsce ładujemy do tablicy okna kolejny punkt startowy. Dzięki temu, przez większość czasu obliczeń, wykorzystujemy pełny zysk z metody Montgomeryego.

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 *tailing effect*.



Rys. 4.1. Schemat działania pojedynczego kroku w pętli

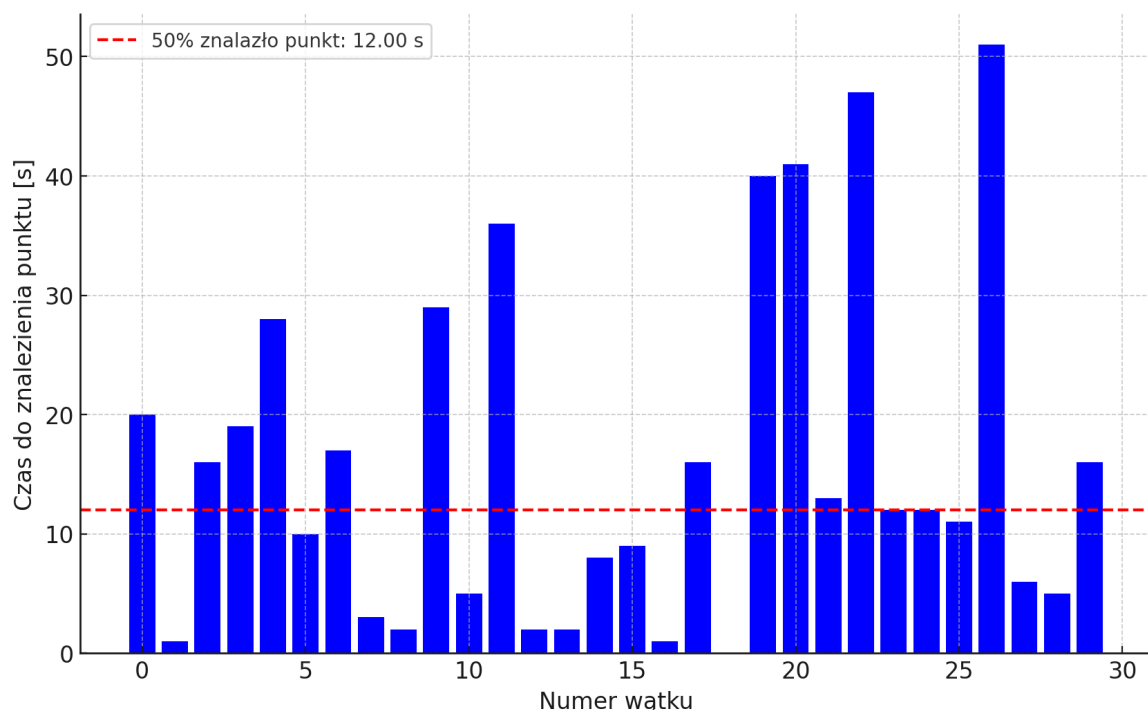
4.6. Tail effect

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

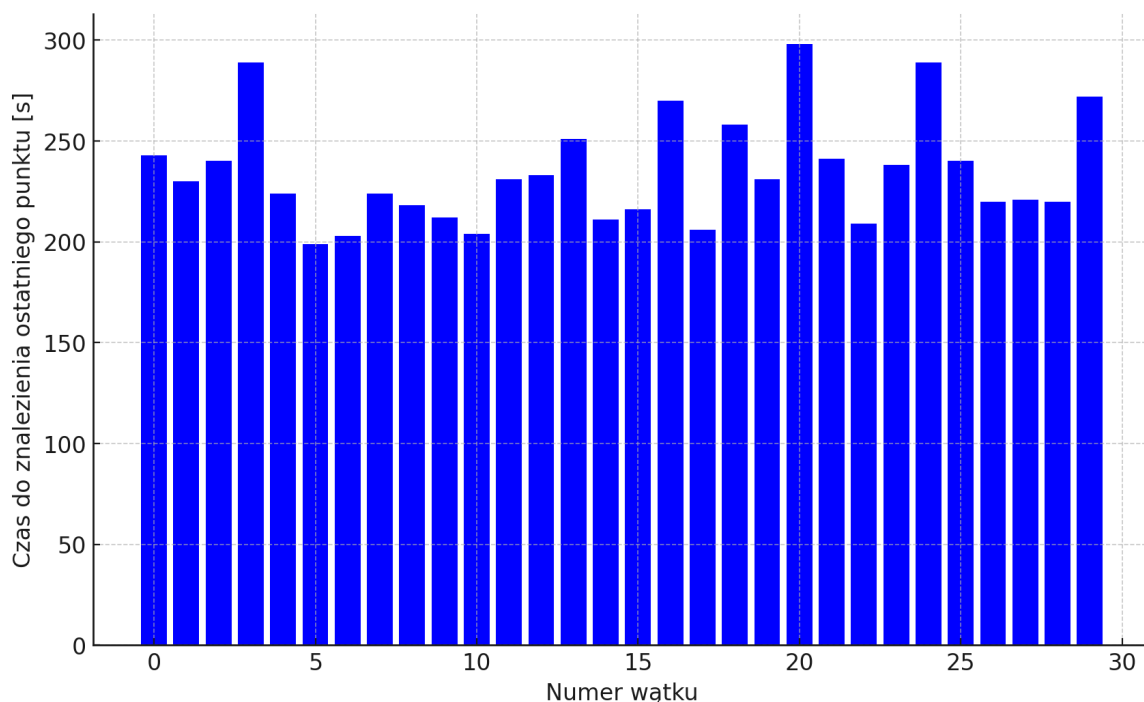
Po znalezieniu punktu wyróżnionego, wątek musi poczekać, na zakończenie pracy wszystkich pozostałych wątków w ramach jednego bloku. To powoduje, że tylko przez krótki czas są wykorzystywane wszystkie zasoby GPU.

Jednym ze sposobów na zmniejszenie tego efektu, jest wydłużenie czasu pracy każdego z wątków poprzez zwiększenie ilości 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 użycie zasobów przez większość czasu obliczeń.

Rys 4.2 oraz 4.4 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. Zmniejsza to wpływ anomalii, gdy punkt wyróżniony jest znajdowany znacznie wcześniej lub później niż wynika z wartości oczekiwanej.



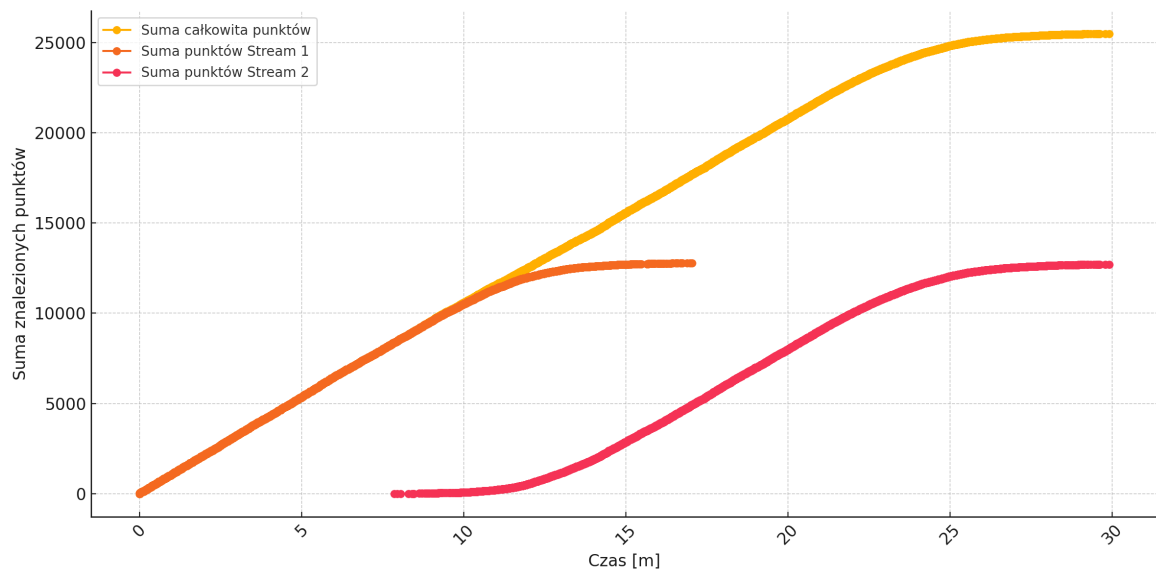
Rys. 4.2. Tail effect widoczny na małej grupie wątków. Wątki kończą pracę po znalezieniu jednego punktu wyróżnionego. Większość czasu obliczeń, jest wykorzystywana tylko przez małą liczbę wątków.



Rys. 4.3. Tail effect widoczny na małej grupie wątków. Wątki kończą pracę po znalezieniu 3 punktów wyróżnionych. Widoczne jest znacznie lepsze wykorzystanie zasobów GPU, przez większość czasu obliczeń.

Aby jeszcze bardziej zniwelować efekt, zastosowałem specjalną flagę 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 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 z kolejki zadań, gdy tylko zwolni się miejsce na kolejny blok, zamiast czekać ze startem na zakończenie poprzedniego kernel'a [**cuda**].

Rys 4.4 przedstawia wykres sumy znalezionych punktów od czasu przy zakolejkowaniu dwóch kernel'i w oddzielnych strumieniach. Widoczne jest przejmowanie przez strumień Stream 2 wolnych zasobów, zwalnianych przez Stream 1. Wypłaszczenie 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 strumień, gdzie 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.

4.7. Program serwera

Główny cel serwera centralnego w równoległej wersji algorytmu Rho Pollarda, sprowadza się do przechowywania obliczonych punktów wyróżnionych oraz poszukiwania wśród nich kolizji. W mojej implementacji, 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 ich kolejekowanie uruchomionych kerneli 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. Do uruchamiania skompilowanego kodu dla GPU, wykorzystałem bibliotekę *Ctypes*, która umożliwia wywoływanie funkcji napisanych w języku C z poziomu Pythona.

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

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 dla tego zastosowania, co umożliwi szybszą i efektywną generację punktów.

Proces generacji punktów startowych odbywa się wyłącznie po stronie serwera. 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. Dzięki temu zasoby są lepiej wykorzystywane, a redundancja w obliczeniach jest zredukowana.

Przykładowy kod generacji punktów w języku Python:

Wydruk 4.4. Generacja punktów startowych

```
def generate_starting_points(instances , zeros_count):
    distinguish_points = []
    starting_points = []
    i = 0
    while i < instances:
        seed = int.from_bytes(random.randbytes(10), "big") % curve_order
        A = P * seed
        x = int(A[0])
        y = int(A[1])
        if not is_distinguish(x, zeros_count):
            i += 1
            starting_points.append((x, y, seed))
        else:
            distinguish_points.append((x, y, seed))
    return starting_points , distinguish_points
```

4.7.1. 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 w algorytmie Rho Pollarda), 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).

Przykładowa implementacja funkcji obliczającej współczynniki a i b w Pythonie:

Wydruk 4.5. Obliczanie współczynników a i b

```
def calculate_ab(seed, precomputed_points: list[PrecomputedPoint]):
    a_sum = seed
    b_sum = 0
    W = P * seed
    while not is_distinguish(W[0], ZEROS_COUNT):
        precomp_index = map_to_index(W[0])
        precomputed = precomputed_points[precomp_index]
        R = precomputed.point
        a_sum = a_sum + precomputed.a
        b_sum = b_sum + precomputed.b
        W = W + R
    a_sum = a_sum % curve_order
    b_sum = b_sum % curve_order
    return (a_sum, b_sum)
```

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 głównym kernelu kodu 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 zakończonych obliczeń w kernelu 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 wykorzystano środowiska PyTest oraz SageMath do przeprowadzenia testów funkcjonalnych, weryfikujących poprawność operacji na krzywych eliptycznych oraz pierwszych kroków algorytmu Rho Pollarda.

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 Pollarda: 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 znalezionej punktu wyróżnionego. Wydajność mierzono na podstawie przyrostu liczby znalezionych punktów w czasie, co pozwalało na oszacowanie liczby iteracji dodawania punktów na sekundę.

Testy przeprowadzono przy założeniu, że punkt wyróżniony spełnia warunek 20 najmłodszych bitów współrzędnej x równych 0 w reprezentacji bitowej. Sprawdzano sumę znalezionych punktów wyróżnionych 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
5 minut	26310	87.7

Na podstawie wyników testów oszacowano liczbę operacji dodawania punktów na sekundę w algorytmie Rho Pollarda. 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, oszacowano, 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 zwerifikowaniu poprawności implementacji na znacznie mniejszych rozmiarach krzywych eliptycznych, przystąpiłem do przeprowadzenia obliczeń dla docelowego problemu. Proces ten wymagał wykonania dużej liczby iteracji algorytmu Rho Pollarda, a jego wyniki zostały szczegółowo udokumentowane.

```
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 4333m16.898s
Wed Sep 18 10:08:40 PM CEST 2024 sys 1m54.288s
```

Rys. 5.1. Wynik działania programu


Poniżej przedstawiono 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

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 publiczny zgodny z założeniami challenge'u Certicom ECCp79.



Type some Sage code below and press Evaluate.

```

15 F = GF(field_order)
16 E = EllipticCurve(F, [curve_a, curve_b])
17 P = E(P_x, P_y)
18 Q = E(Q_x, Q_y)
19 result = 92221507219705345685350
20
21 print(f"P coords: {P}")
22 print(f"Q coords: {Q}")
23 print(f"n*P coords: {result * P}")
24 print(result*P == Q)

```

Evaluate Language: Sage

Share

```

P coords: (233116918217145689900413 : 15924004798469701256458 : 1)
Q coords: (30575716130623743443813 : 304679994529239484325803 : 1)
n*P coords: (30575716130623743443813 : 304679994529239484325803 : 1)
True

```

[Help](#) | Powered by [SageMath](#)

Rys. 5.2. Weryfikacja wyniku w SageMath

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

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. Oczekiwana 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.

6. Porównanie wyników

Praca najbardziej zbliżona tematycznie do mojej to *Solving prime-field ECDLPs on GPUs with OpenCL* autorstwa Erika Bossa [7]. W odróżnieniu od mojej pracy, implementacja wykorzystana przez Bossa 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.

Wyniki mojej pracy, zrealizowanej w 2024 roku przy użyciu karty NVIDIA GTX 2070 Super, pokazują wydajność na poziomie 87,7 milionów operacji na sekundę. Dla porównania, praca Erika Bossa z 2015 roku osiągnęła wydajność 109 milionów iteracji na sekundę dla karty AMD. Mimo niewielkiej różnicy, moja implementacja osiąga wynik, zbliżony rzędem wielkości do wyniku z tej pracy. Wyniki uzyskane przez Bossa wskazują, że obliczenie problemu logarytmu dyskretnego dla krzywej 112-bitowej zajęłoby odpowiednio 19,81 lat dla karty AMD i 31,77 lat dla karty NVIDIA.

Na podstawie wyników uzyskanych w mojej pracy, ekstrapolując wydajność z krzywej 79-bitowej, rozwiązanie ECDLP dla krzywej 112-bitowej zajęłoby około 32 lat na GPU NVIDIA GTX 2070 Super. Więc w przypadku kart graficznych Nvidia, oba wyniki są prawie identyczne, pomimo nieznacznie nowszej wersji w przypadku mojej karty graficznej.

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

Różnica może wynikać z większej optymalizacji pod konkretną kartę graficzną w przypadku pracy Boss'a oraz zastosowania techniki *Negation Map* która pozwala przyspieszyć czas obliczeń o współczynnik $\sqrt{2}$ [3].

Wiele pozostałych prac o podobnej tematyce, wykorzystuje układy FPGA do obliczania logarytmu dyskretnego. Jednak zazwyczaj dotyczą one krzywej eliptycznej na ciele binarnym [30]. W takim przypadku, porównanie traci na wartości ponieważ na ciele binarnym można zastosować znacznie więcej optymalizacji przy operacjach na krzywej.

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, s. 236. 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] 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.
- [9] John Cheng, Max Grossman i Ty McKercher. *Professional CUDA C Programming*. Wrox Press, 2014, s. 528. ISBN: 9781118739273.
- [10] Andrzej Chrzęszczczyk. *Algorytmy teorii liczb i kryptografii w przykładach*. 2010, s. 328. ISBN: 9788360233672.
- [11] *CUDA C++ Programming Guide*. <https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>. Accessed: 2024-09-30.
- [12] 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.
- [13] 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.
- [14] 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.
- [15] 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.

- [16] Ryan Henry i Ian Goldberg. „Solving Discrete Logarithms in Smooth-Order Groups with CUDA 1”. W: (). URL: <http://cacr.uwaterloo.ca/>.
- [17] Sanders Jason i Kandrot Edward. *Cuda by example*. Addison-Wesley Professional, 2011, s. 313. ISBN: 9786612660122.
- [18] 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.
- [19] 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.
- [20] 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/>.
- [21] Alfred J. Menezes, Paul C. Van Oorshot i Scott A Vanstone. „Handbook of Applied Cryptography”. W: (2001).
- [22] 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>.
- [23] Paul C Van Oorschot i Michael J Wiener. *Parallel Collision Search with Cryptanalytic Applications*. 1999.
- [24] 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.
- [25] J M Pollard. „Monte Carlo Methods for Index Computation (mod p)”. W: 32 (143 1978), s. 918–924.
- [26] 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.
- [27] Douglas R. Stinson i Maura B. Paterson. *Kryptografia. W teorii i praktyce*. IV. 2021.
- [28] Edlyn Teske. „ON RANDOM WALKS FOR POLLARD’S RHO METHOD”. W: *MATHEMATICS OF COMPUTATION* 70 (234 2000).
- [29] 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.

- [30] 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 Gates Array

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	12
2.2. $P + Q$ na krzywej eliptycznej $y^2 + y = x^3 - x^2 + 2x$	13
2.3. Krzywa eliptyczna $y^2 = x^3 - 4x + 2$ nad $GF(2^{11} - 9)$	14
3.1. Schemat architektury	20
4.1. Schemat działania pojedynczego kroku w pętli	32
4.2. Tail effect widoczny na małej grupie wątków. Wątki kończą pracę po znalezieniu jednego punktu wyróżnionego. Większość czasu obliczeń, jest wykorzystywana tylko przez małą liczbę wątków.	33
4.3. Tail effect widoczny na małej grupie wątków. Wątki kończą pracę po znalezieniu 3 punktów wyróżnionych. Widoczne jest znacznie lepsze wykorzystanie zasobów GPU, przez większość czasu obliczeń.	34
4.4. Suma znalezionych punktów wyróżnionych z podziałem na strumienie, gdzie 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.	35
5.1. Wynik działania programu	40
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	Prototypy funkcji wykorzystywanych do operacji na długich liczbach	27
4.2	Inicjalizacja pamięci współdzielonej i flagi <code>warp_finished</code>	28
4.3	Funkcja przydzielająca punkt	29
4.4	Generacja punktów startowych	36
4.5	Obliczanie współczynników a i b	37

Spis załączników