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 2024

Akceleracja sprzętowa kryptoanalizy algorytmów kryptograficznych

Streszczenie. Cel pracy

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

Hardware acceleration of cryptoanalysis of cryptograhpic algorithms

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

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

załącznik nr 3 do zarządzenia
nr 28 /2016 Rektora PW

Politechnika Warszawska

……..
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

DO UZUPEŁNIENIA - rsa krzywych rozmiary klucza kryptoanaliza do testowania bezpieczenstwa krzywych

1.1. Cel pracy

Celem tej pracy jest realizacja systemu korzystającego z koprocesora GPU, w celu przyśpieszenia obliczeń przy rozwiązywaniu problemu logarytmu dyskretnego na krzywej eliptycznej ECCp-79 z challange'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.

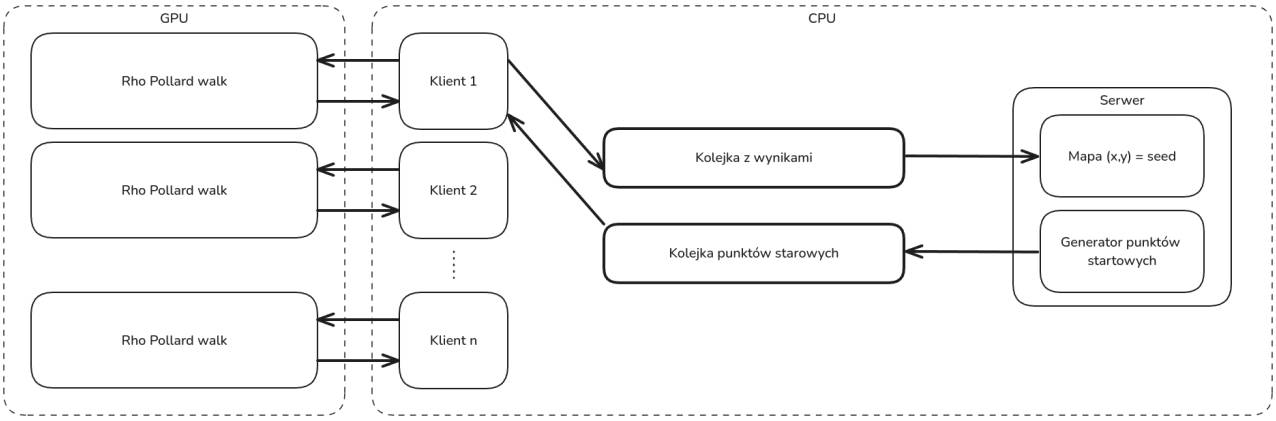
1.2. Koncepcja

Projekt zakłada zaimplemntowanie systemu do obliczania logarytmu dyskretnego na krzywej eliptycznej. W celu akceleracji obliczeń wykorzystuje koprocesor jakim jest GPU. Ponieważ równoległy algorytm Rho Pollarda zakłada architekturę w postaci klient-serwer, to na projekt składa się program pełniący rolę centralnego serwera, oraz program kliencki wykonujący obliczenia na GPU, uruchamiany w wielu instancjach.

Zarówno program serwera jak i klienta, działają w ramach jednego PC, jednak architektura pozwala na wykorzystanie wiecej niż jednej karty graficznej w celu przyspieszenia obliczeń. Program serwera jest zaimplementowany z wykorzystaniem języka Python i odpowiada za gromadzenie znalezionych punktów oraz za generację nowych punktów startowych. W celu optymalnego przechowywania punktów, wykorzystuje hash mapę w formie: (*współrzędne punktu* : *seed*), która pozwala na szybkie sprawdzanie czy dany punkt już został znaleziony, oraz porównanie ziarna użytego do wygenerowania punktu początkowego.

Część klienta odpowiedzialna za komunikację z serwerem, również jest zaimplementowana w języku Python i za pomocą interfejsu ABI zleca obliczenia do programu napisanego z wykorzystaniem CUDA C++. Zarówno program serwera jak i każdego z działających klientów, uruchamiany jest w osobnym wątku, co pozwala współbieżne szukanie kolizji i efektywne kolejkowanie kolejnych serii obliczeń na GPU.

W celu komunikacji klientów z serwerem, wykorzystywane są asynchroniczne kolejki z biblioteki standardowej Pythona.



Rys. 1.1. Schemat architektury

2. Wstęp teoretyczny

2.1. Procesor graficzny

DO UZUPEŁNIENIA GŁÓWNIE SŁOWNICTWO oraz architektura SIMD SM - streaming multiprocesor BLOCK - grupa wątków, najbardziej granularny sposób rozpoczynania obliczeń wątek - pojedyncza logiczna jednostka obliczeniowa, wykonuje instrukcje w grupach po 32 (rozgałęzienia powodują sekwencyjność w tej grupie 32) good practice: brak rozgałęzień w kodzie, jedna instrukcja, wiele danych

Hierarchia pamięci

2.2. Krzywe eliptyczne

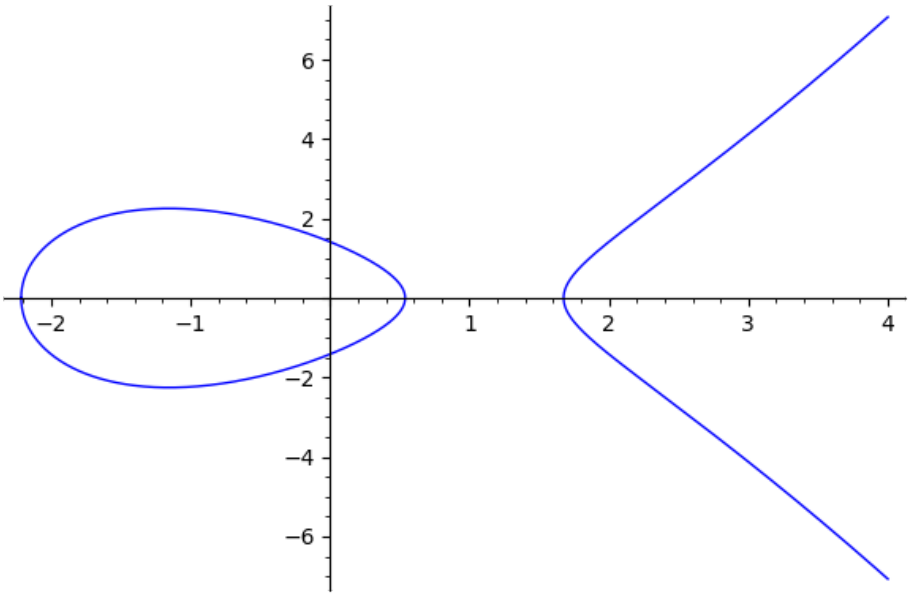
Zakładając, że ciało $\mathbb{K}$ ma charakterystykę różną od 2 i 3, oraz że stałe $a, b \in \mathbb{K}$ spełniają warunek:

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

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 [10].



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

Krzywe eliptyczne zdefiniowane na liczbach rzeczywistych nie są kluczowe w systemach kryptograficznych [10], ale takie ustawienia pozwalają na prostsze przedstawienie niektórych zagadnień np. dodawnie punktów na krzywej.

2.2.1. Dodawanie punktów na krzywej eliptycznej

Odpowiednie zdefiniowanie operacji dodawania punktów na krzywej eliptycznej pozwala otrzymać grupę abelowa, 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 prostopadłą linią, która przecina krzywą w trzecim punkcie, R'. Następnie, wynikowy punkt R, będący sumą $P + QP + Q$, 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 $2P$ [4][10].

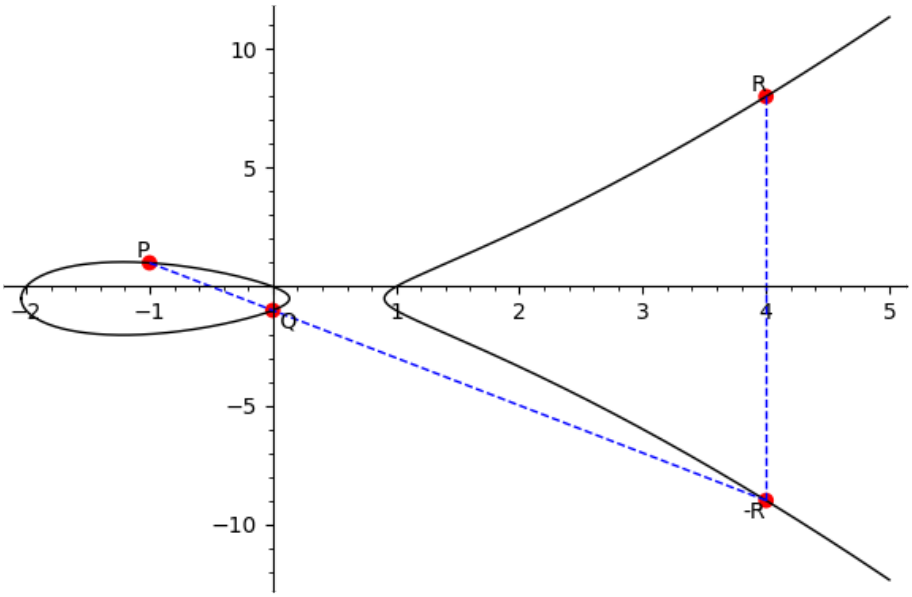
Definując dodawanie punktów na krzywej eliptycznej w sposób algebraiczny otrzymujemy następujące wzory:

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

$$\lambda = \frac{y_2 - y_1}{x_2 - x_1}, \tag{1}$$

$$x_3 = \lambda^2 - x_1 - x_2, \tag{2}$$

$$y_3 = \lambda(x_1 - x_3) - y_1 \tag{3}$$



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 $P = -Q$:

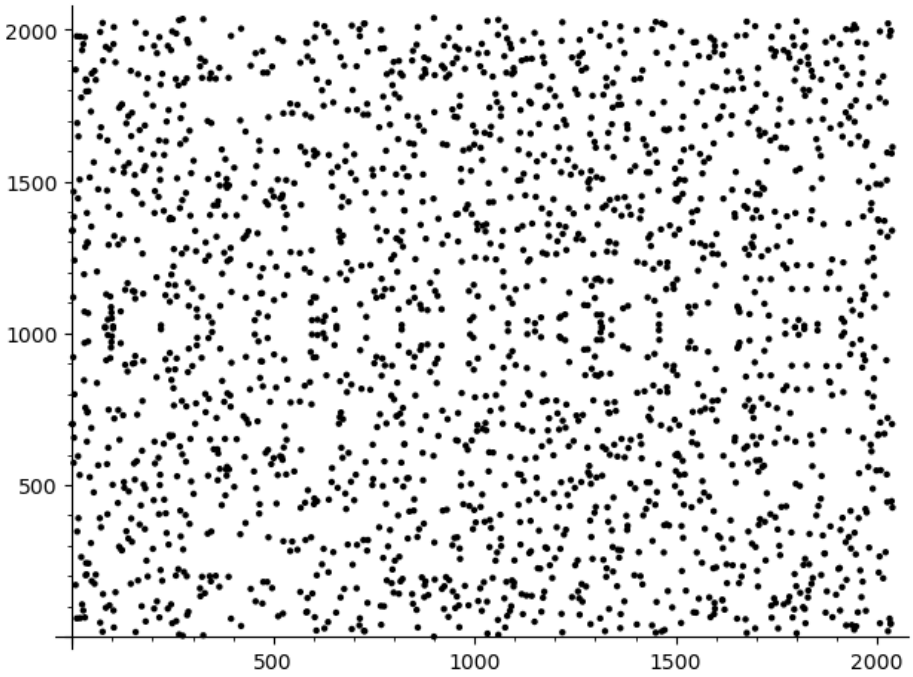
$$P + (-P) = \mathcal{O}$$

Dodatkowo odwrotność punktu na krzywej P definujemy jako $-P = (x, -y)$ [10].

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 $y^2 = x^3 - 4 + 2$ 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[10].

2.2.4. Problem logarytmu dyskretnego na krzywej eliptycznej

W przypadku krypografii 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:

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

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 [10].

2.3. Algorytm Rho Pollarda

RYSUNEK LITERKI RHO (bo jak inaczej bez tego mówic o rho pollardzie :))

DODAC OPIS, co się dzieje po znalezieniu kolizji, jak to pozwala obliczyć logarytm dyskretny

CZY TRZEBA opisywać algorym sekwencyjny skoro i tak korzystam z równoległego? Moze samo odesłanie do literatury

DO UZUPEŁNIENIA

Najszybszym znanym algorytmem rozwiązującym problem logarytmu dyskretnego na krzywej eliptycznej jest algorytm Rho Pollarda, zaproponowany przez Johna Pollarda w 1978 roku [9]. Pozwala on na znalezienie logarytmu dyskretnego w czasie $O(\sqrt{n})$, jednak jest to jedynie czas *oczekiwany*, ponieważ ze względu na losową naturę algorytmu [2]. W porównaniu do innego znanego algorytmu, Baby-Step Giant-Step [10], algorytm Rho Pollarda jest bardziej efektywny pamięciowo, nie wymagając przestrzeni $O(\sqrt{n})$ a jedynie $O(1)$ w wersji sekwencyjnej [10][2].

Klasyczny algorytm Rho Pollarda, oparty o poszukiwanie cyklu, słabo się skaluje w przypadku zrównoleglenia, osiągając jedynie przyśpieszenie rzędu $O(\sqrt{m})$ dla m procesorów [5]. Dlatego w swojej pracy wykorzystałem równoległą wersję algorytmu, zaproponowaną przez Van Oorschota i Wienera [8].

2.3.1. Równoległy algorytm Rho Pollarda

Równoległa wersja algorytmu Rho Pollarda, zakłada zastosowanie wielu równoległych ścieżek błądzenia.

Jednostki obliczeniowe poszukują wtedy specalnych punktów nazywanych wyróżnionymi, które następnie są przekazywane do centralnego serwera w celu znalezienia kolizji między nimi.

Ponieważ sprawdzenie, czy punkt jest wyróżniony następuje w każdej iteracji, ważne jest aby kryterium decydujące o uznaniu punktu za wyróżniony, było możliwie szybkie do sprawdzenia. Często stosowaną metodą, jest sprawdzanie ilości zer na początku lub na końcu binarnej reprezentacji współrzędnej x.

Adding walk zakłada jedynie wykonywanie kolejnych operacji dodawania punktów.

Niech W_0 będzie punktem startowym, a f funkcją haszującą $f : \langle P \rangle \rightarrow \{1, ..s\}$ o możliwie jednostajnym rozkładzie. Następnie, potrzebujemy tablicy wstępnie obliczonych punktów: $R_i = c_i P + d_i Q$ dla $0 \leq i \leq s-1$. Funkcja iteracyjna jest wtedy zdefiniowana w następujący sposób:

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

Ważną kwestią jest też rozmiar tablicy z punktami wstepnie obliczonymi. Zbyt mały rozmiar powoduje, że funkcja nie będzie dostatecznie losowa. W eksperymentach praktycznych pokazano, że dla $s \geq 16$, funkcja zapewnia wystarczający poziom losowości, niezależnie od rozmiaru grupy. [11]

Istotną zaletą tej funkcji w przypadku programu uruchamianego na GPU, jest uniknięcie rozgałęzień podczas każdej iteracji, co jest szczególnie istotne w przypadku architektury SIMD. Prawie zawsze wykonujemy tą samą operację dodawania dwóch punktów, poza mało prawdopodobnym przypadkiem w którym $W_i == R_{f(W_i)}$.

3. 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 częśći klienta napisanej w CUDA C++, poza sekcją na końcu, poświęconą serwerowi oraz częsci klienta działającej na CPU.

3.1. Narzędzia oraz sprzęt

3.1.1. Sprzęt

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

3.1.2. 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.

3.1.3. System budowania oraz kompilacja CUDA C++

W celu zarządzania kompilacją częsci 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 wesji 13.3.1.

3.1.4. Testy

Wszelkie testy poprawności implemetnacji 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 przyśpieszyło 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.

3.2. Artmetyka na ciele F_p

3.2.1. Reprezentacja długich liczb

Natywnie w języku C oraz CUDA C++ nie istnieje typ danych, który pozwalałby na przechowywanie liczb większych niż 64 bitowe. W celu przeprowadzenia obliczeń na liczbach większych niż natywny rozmiar słowa bitowego, niezbędna jest odpowiednia implementacja nowych typów danych jak i samych operacji. W przypadku ciała F_p, gdzie p jest liczbą pierwszą o rozmiarze 79 bit, potrzebujemy conajmniej 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:

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

Zaimplemetowany jako tablica typu u_int32:

```
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 wspórzędnych daje oszczędnośc $2 \cdot 2 \cdot 32 = 128$ bitów na punkt znajdujący się w pamięci.

3.2.2. Operacje na długich liczbach

Do operacji na długich liczbach, odpowiednio dostosowałem małą bibliotekę dostępną w domenie publicznej: *tiny-bingum-c*. Bilbioteka ta dostarcza podstawowe operacje na dużych liczbach w postaci wektorów, takie jak dodawanie, odejmowanie, mnożenie czy dzielenie. Wykorzystuje w tym celu standardowe algorytmy [6] wykorzystywane przy obliczeniach *multiple precision*. Dużą zaletą ten biblioteki jest jej prostota i brak wykorzysywania standardowej biblioteki C oraz dynamicznej alokacji pamięci. Wszystkie operacja są wykonywane z wykorzystaniem stosu, co pozwala na jej wykorzystanie w środowiskach takich jak GPU, gdzie dostęp do dynamicznej alokacji pamieci 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.

3.2.3. Dodawanie i odejmowanie modulo

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

W przypadku odejmowanie dówch 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 $a + b >= p$ zredukować wynik odejmując od niego liczbę p.

3.2.4. 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.

3.2.5. 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.

3.2.6. 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 dodawnia, odejmowanie czy mnożenie nawet do 32 tyś. 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*.

Algorithm 1 Odwrotność modularna a mod b

Input: a, b
$b_0 \leftarrow b$
$x_0 \leftarrow 0$
$x_1 \leftarrow 1$
$x_{0\_sign} \leftarrow 0$
$x_{1\_sign} \leftarrow 0$
while $a > 1$ **do**
    $q \leftarrow a \div b$
    $t \leftarrow b$
    $b \leftarrow a \mod b$
    $a \leftarrow t$
    $t_2 \leftarrow x_0$
    $t_{2\_sign} \leftarrow x_{0\_sign}$
    $qx_0 \leftarrow q \times x_0$
    if $x_{0\_sign} \neq x_{1\_sign}$ **then**
        $x_0 \leftarrow x_1 + qx_0$
        $x_{0\_sign} \leftarrow x_{1\_sign}$
    else
        if $x_1 > qx_0$ **then**
            $x_0 \leftarrow x_1 - qx_0$
            $x_{0\_sign} \leftarrow x_{1\_sign}$
        else
            $x_0 \leftarrow qx_0 - x_1$
            $x_{0\_sign} \leftarrow 1 - x_{0\_sign}$
        end if
    end if
    $x_1 \leftarrow t_2$
    $x_{1\_sign} \leftarrow t_{2\_sign}$
end while
if $x_{1\_sign} == 1$ **then**
    return $b - x_1$
else
    return x_1
end if

3.3. Funkcja iterująca

DO UZUPEŁNIENIA - max 0.5 strony - ogólnie przypomnienie tego co było na początku w teorii, że addition walk jest lepszy dla SIMD niż klasyczna wersja rho pollarda

3.3.1. 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 [**CUDA_BOOK**]. Ponieważ ilość punktów która zapewnia dostatecznie losowy spacer po krzywej eliptycznej jest stosunkowo niewielka [11], 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 ze współrzędną 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.

3.3.2. 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.

3.3.3. Dodawanie punktów na krzywej eliptycznej

DO UZUPEŁNIENIA - max 0.5 strony większość jest poniżej. MOZE NAWET POMINĄC, w sumie to przepisanie tego co jest w teorii z wykorzystaniem zaimplementowanych prymitywów matematycznych???

3.3.4. Obliczanie odwrotnosci 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 [2].

Koszt dodawania punktów na krzywej:

$$1O + 3M$$

Najdroższym działaniem, które wpływa na wysoki koszt obliczeniowy mnożenia modularnego oraz odwrotności w ciele, jest operacja dzielenia [6]. 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 przyśpieszyć obliczenia, zastosowałem technikę obliczania wielu odwrotności za jednym razem, znaną jako *Montgomery Trick* [7].

Idea stojąca za tym sposobem, jest następująca. Niech $x_1, \cdots, x_n$ będą elementami, których odwrotność chcemy policzyć. Na początku obliczamy tablicę elementów, w postaci $a_1 = x_1, a_2 = x_1 \cdot x_2, \cdots, a_n = x_1 \cdots x_n$. 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 tej metody wymagał podjęcia paru decyzji. Pierwszym problemem który pojawia się w metodzie Montgomerego, 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 [3], 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 pamięc w postaci rejestrów. Dodatkowo, z racji, że tylko jeden wątek oblicza odwrotności, pozostałe muszą bezczynnie 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 Montgomerego, 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:

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

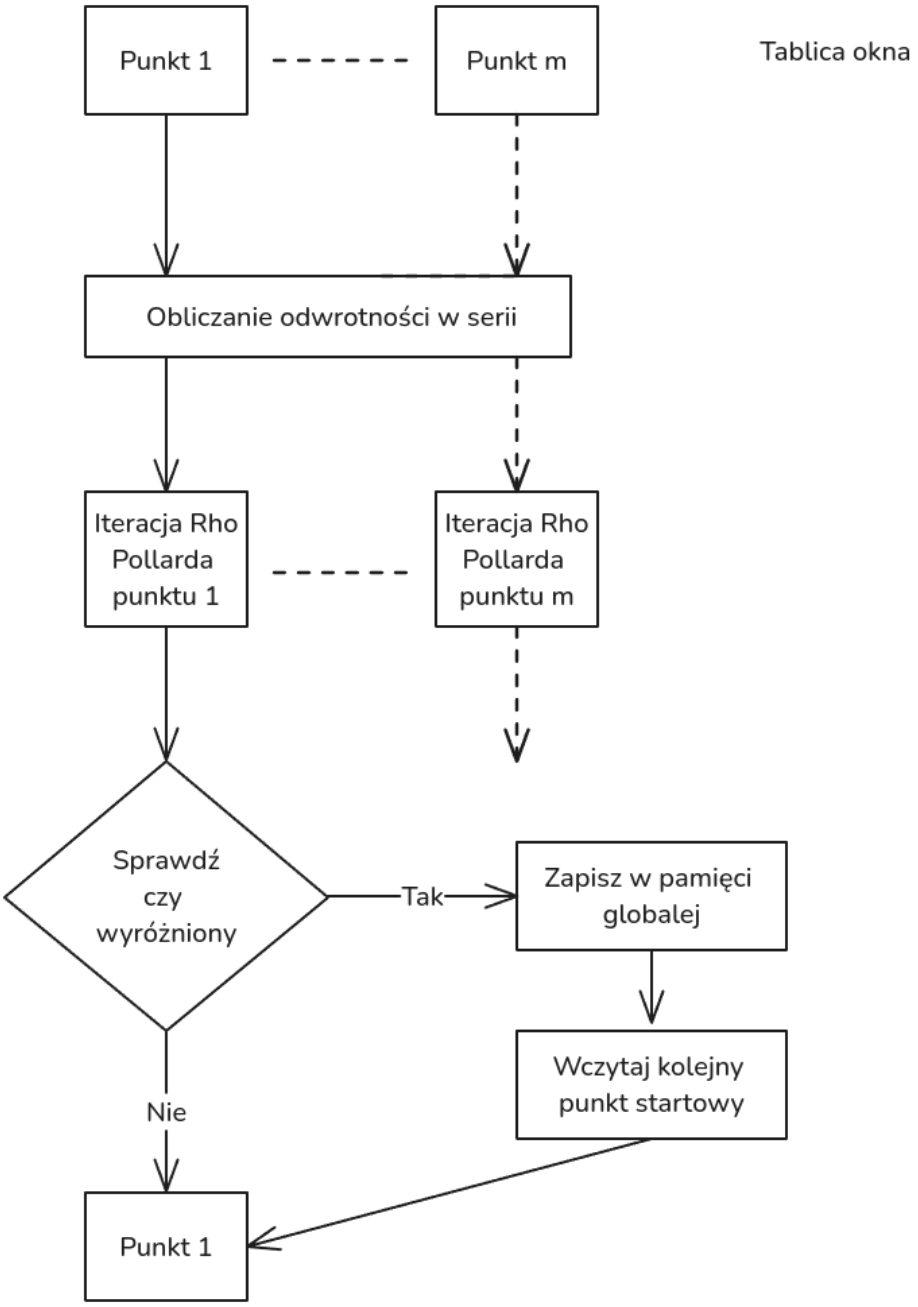
Gdzie $3(i)M$ to mnożenia, które nie przyczyniają się do znalezienia nowych punktów wyróżnionych, więc należy je traktować jako niepotrzebne spowolnienie.

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 dodawnia 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 każdy z nowo obliczonych punktów, czy 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 Montgomerego.

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*.

Wzrost wydajności mierzony w ilości operacji dodawania punktów na krzywej na sekundę, który udało mi się osiągnąć, wynosi około 90% względem wersji, która nie wykorzystuje metody Montgomerego.



Rys. 3.1. Schemat działania pojedyńczego kroku w pętli z wykorzystaniem okienkowania

3.4. 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

Wielkość okna	Średnia ilość operacji na sekundę	Wzrost wydajności (%)
Brak	1000	0%
5	1500	25%
10	1700	30%
15	1900	35%
20	2100	40%

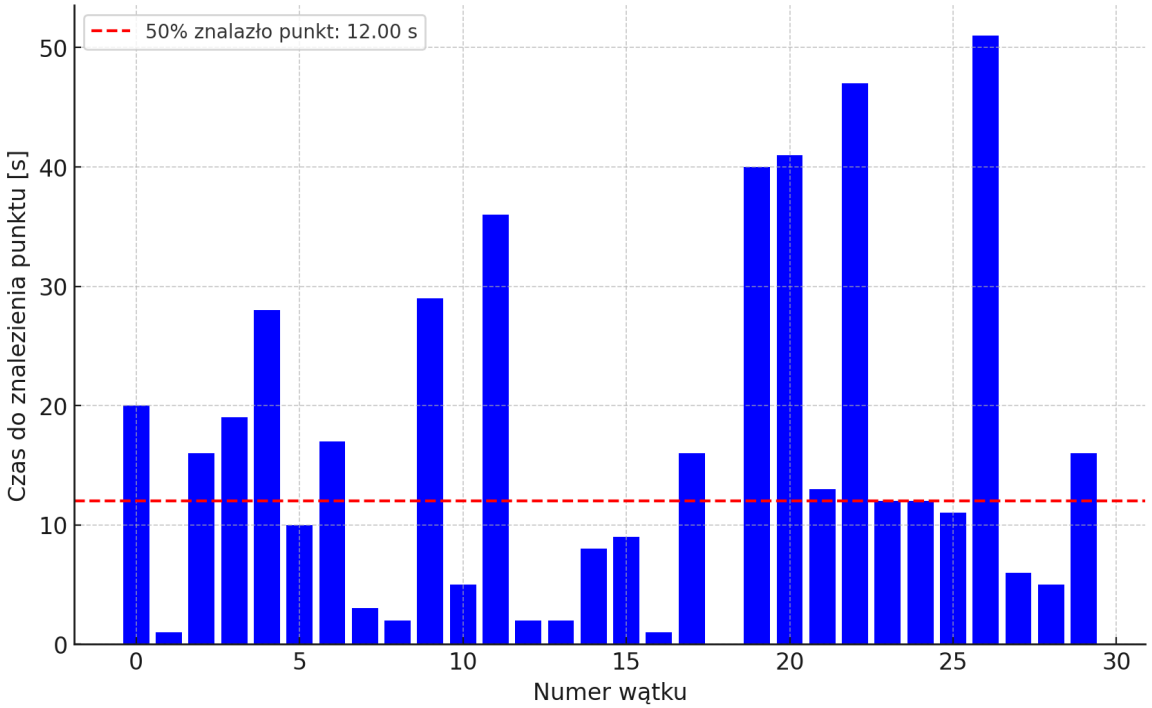
Tabela 3.1. Porównanie algorytmu z metodą Montgomery DANE NA RAZIE W FORMIE PLACEHOLDERA, NIE ZDAZYLEM PRZESZUKAC LOGOW W POSZUKIWANIU DOKLADNYCH WYNIKOW, ale coś około 96 milionów vs 56 milionów dla okna rozmiaru 10

widoczne, ze względu na losowość funkcji błądzenia po krzywej. Wątki znajdują punkty wyróżnione w różnym, losowym czasie.

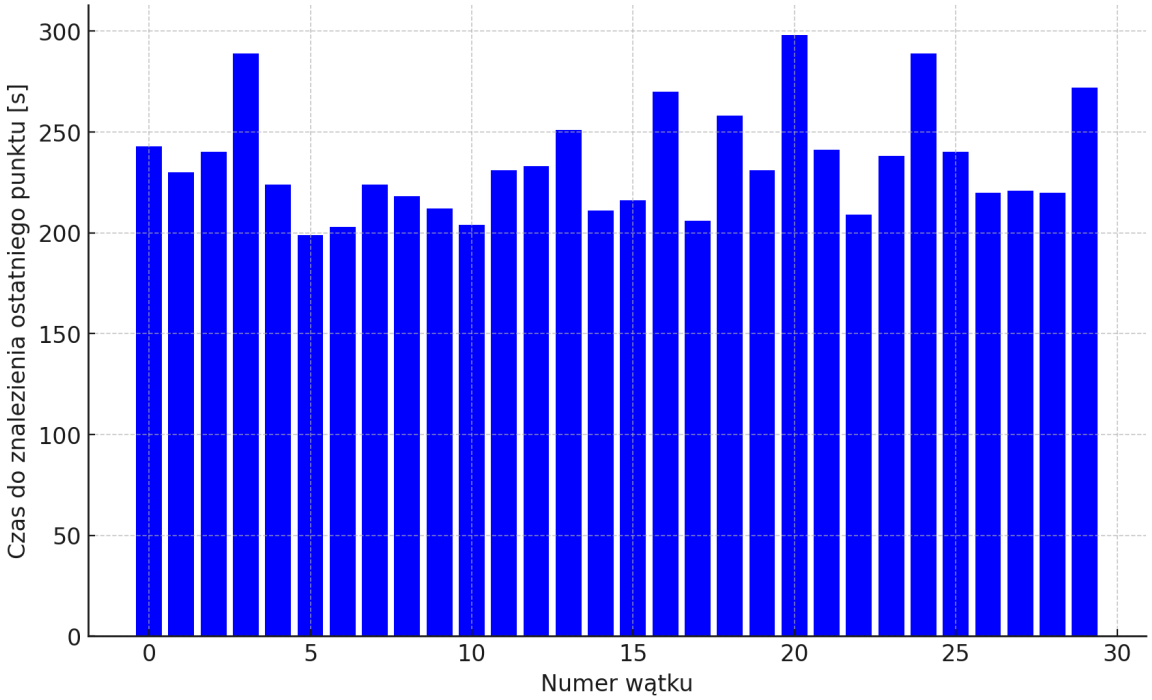
Po znalezieniu punktu wyróżnionego, wątek musi zaczekać, na zakończenie pracy wszystkich pozostałych wątków w ramach jednego bloku. To powoduje, że tylko przez krótki czas są wykorzystywane wszystkie zasoby GPU.

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 utylizację zasobów przez większość czasu obliczeń.

Rys 3.2 oraz 3.4 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 znaczne 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.



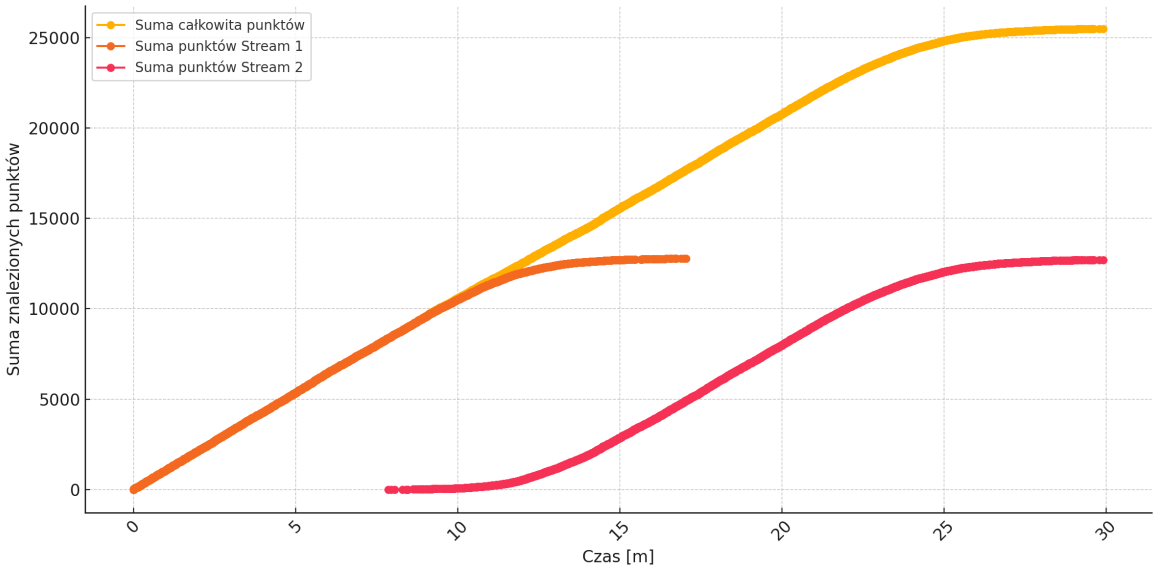
Rys. 3.2. Tail effect widoczny na małej grupie wątków. Wątki kończą pracę po znalezieniu jednego punktu wyróżnionego. Większość czasu obliczeń, jest wykorzystywana tylko przez małą liczbę wątków.



Rys. 3.3. 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 kolejkowanie 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 3.4 przedstawia wykres sumy znalezionych punków od czasu przy zakolejkowaniu dwóch kernel'i w odzielnych 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. 3.4. 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.

3.5. Serwer

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 rozszczerzone 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 kolejkowanie 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

DO UZUPEŁNIENIA stosuję metodę generacji za pomocą MD5 modulo rząd ciała

3.5.1. Kolizje

DO UZUPEŁNIENIA Opisać co się dzieje w przypadku kolizji, jak odtwarzana jest sciezka bladzenia przypomniec jak z tego obliczyc logarytm dyskretyn -> ALGORYTM RHO POLLARD w sekcji teorii bedzie miał wyjasnione

4. Wyniki

DO UZUPELNIENIA

4.1. Dalsze usprawnienia

Redukcja berreta Zastosowanie pamięci do tekstur

4.2. Porównanie z innymi pracami

DO UZUPELNIENIA

Bibliografia

[1] Daniel J Bernstein i in. „ECC2K-130 on NVIDIA GPUs”. W: (2012). URL: http://www.ecc-challenge.info.

[2] Ian F Blake, Gadiel Seroussi i Nigel Paul Smart. *Krzywe eliptyczne w kryptografii*. 2005, s. 236. ISBN: 9788320429510.

[3] Erik Boss. *Solving prime-field ECDLPs on GPUs with OpenCL*. 2015.

[4] Andrzej Chrzęszczyk. *Algorytmy teorii liczb i kryptografii w przykladach*. 2010, s. 328. ISBN: 9788360233672.

[5] Ryan Henry i Ian Goldberg. „Solving Discrete Logarithms in Smooth-Order Groups with CUDA 1”. W: (). URL: http://cacr.uwaterloo.ca/.

[6] Alfred J. Menezes, Paul C. Van Oorshot i Scott A Vanstone. „Handbook of Applied Cryptography”. W: (2001).

[7] 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.

[8] Paul C Van Oorschot i Michael J Wiener. *Parallel Collision Search with Cryptanalytic Applications*. 1999.

[9] J M Pollard. „Monte Carlo Methods for Index Computation (mod p)”. W: 32 (143 1978), s. 918–924.

[10] Douglas R. Stinson i Maura B. Paterson. *Kryptografia. W teorii i praktyce*. IV. 2021.

[11] Edlyn Teske. „ON RANDOM WALKS FOR POLLARD’S RHO METHOD”. W: *MATHEMATICS OF COMPUTATION* 70 (234 2000).

Wykaz symboli i skrótów

EiTI – Wydział Elektroniki i Technik Informacyjnych
PW – Politechnika Warszawska
FPGA – Field Programmable Gates Array
DLP – Discrete Logarithm Problem
GF – Galois Field (ciało skończone)

Spis rysunków

Spis tabel

Spis wydruków

Spis załączników