МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

ФАКУЛЬТЕТ КІБЕРБЕЗПЕКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

Кафедра інформаційних технологій

КВАЛІФІКАЦІЙНА РОБОТА

на тему:

«ПРОГРАМНА РЕАЛІЗАЦІЯ ПРОЦЕСУ ПОШУКУ ПРОСТИХ ЧИСЕЛ ЗАСОБАМИ NVIDIA CUDA»

Рівень «бакалавр»

Галузь знань 12 «Інформаційні технології»,

спеціальність 122 «Комп'ютерні науки»

Виконав здобувач 4 курсу

Чернецький Володимир Андрійович

Керівник к.т.н., доцент

Чикунов Павло Олександрович

Рецензент к.т.н., доцент

Щербина Юрій Володимирович

Одеса – 2024

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «ОДЕСЬКА ЮРИДИЧНА АКАДЕМІЯ»

Факультет кібербезпеки та інформаційних технологій

Кафедра інформаційних технологій

Галузь знань 12 Інформаційні технології

Спеціальність 122 «Комп'ютерні науки»

Назва освітньої програми «Комп’ютерні науки»

ЗАТВЕРДЖУЮ

Завідувач кафедри інформаційних технологій

доцент Наталія Логінова _____

«____ »_____ 2024 року

З А В Д А Н Н Я

на кваліфікаційну (бакалаврську) роботу

здобувачу Чернецькому Володимиру Андрійовичу

1. Тема роботи: «Програмна реалізація процесу пошуку простих чисел засобами NVIDIA CUDA».

Керівник роботи: к.т.н, доцент Чикунов Павло Олександрович

затверджені наказом НУ «ОЮА» від «02» квітня 2024 року № 809-06

2. Строк подання здобувачем роботи «20» травня 2024 року

3. Дата видачі завдання «15» вересня 2023 року

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів бакалаврської роботи	Строк виконання етапів роботи	Примітка
1	Аналіз предметної області та існуючих аналогів	10.01.2023	
2	Формулювання цілей та завдань дослідження	12.02.2023	
3	Розробка алгоритмів пошуку простих чисел з використанням GPU	15.03.2023	
4	Програмна реалізація та тестування CPU- та CUDA-реалізацій	28.04.2024	
5	Оформлення звітної частини	17.05.2024	

Здобувач _____Володимир Чернецький_____

(підпис) (ім’я та прізвище)

Керівник роботи _____Павло Чикунов _____

(підпис) (ім’я та прізвище)

АНОТАЦІЯ

Чернецький В. А. «Програмна реалізація процесу пошуку простих чисел засобами NVIDIA CUDA». – Кваліфікаційна робота бакалавра за спеціальністю 122 «Комп'ютерні науки», освітньою програмою «Комп’ютерні науки».

Кваліфікаційна робота викладена на 51 сторінках основного тексту і 70 сторінках загального тексту, містить 3 розділи, 37 рисунків, 2 таблиці, 2 додатка, 27 використаних джерел.

Метою виконання роботи є проектування, розробка, тестування та оцінка ефективності CUDA-реалізацій різних алгоритмів пошуку простих чисел.

Об'єктом роботи є методи та засоби організації процесу пошуку простих чисел з використанням графічного процесора.

Предметом роботи є техніки розробки та реалізації різних алгоритмів пошуку простих чисел з використанням програмно-апаратної платформи NVIDIA CUDA.

В результаті аналізів результатів тестування та профілювання програмних CPU- та CUDA-реалізацій різних алгоритмів пошуку простих чисел встановлено, що використання CUDA-технології для рішення складних обчислювальних задач, зокрема для пошуку простих чисел за допомогою решета Ератосфена, колісної факторизації та тесту Мілера-Рабіна, значно зменшує час виконання та навантаження на центральний процесор, тому заслуговує подальших досліджень.

Ключові слова: прості числа, решето Ератосфена, колісна факторизація, решето Аткіна, тест Мілера-Рабіна, числа Мерсена, тест Люка-Лемера, Visual Studio, мова C++, NVIDIA CUDA, GPU, CPU.

ABSTRACT

Chernetskyi V. A. "Software Implementation of Prime Number Search Process using NVIDIA CUDA". – Bachelor's Thesis in Computer Science, Specialization 122 "Computer Science", Educational Program "Computer Science".

The thesis consists of 51 pages of main text and 70 pages in total, containing 3 chapters, 37 figures, 2 tables, 2 appendices, and references to 27 sources.

The aim of this work is the design, development, testing, and evaluation of CUDA implementations of various prime number search algorithms.

The object of the work is methods and means of organizing the process of searching for prime numbers using a graphics processor.

The subject of the work is the techniques for developing implementations of various prime number search algorithms using the NVIDIA CUDA software-hardware platform.

Analysis of the testing results and profiling of CPU and CUDA implementations of various prime number search algorithms revealed that the use of CUDA technology for solving complex computational tasks, particularly for prime number search using the Sieve of Eratosthenes, wheel factorization, and Miller-Rabin test, significantly reduces execution time and central processor load. Therefore, it deserves further investigation.

Keywords: prime numbers, Sieve of Eratosthenes, wheel factorization, Sieve of Atkin, Miller-Rabin test, Mersenne numbers, Lucas-Lehmer test, Visual Studio, C++, NVIDIA CUDA, GPU, CPU.

ЗМІСТ

ПЕРЕЛІК СКОРОЧЕНЬ

CPU – Central Processing Unit, центральний процесор.

GPU – Graphics Processing Unit, графічний процесор.

GPGPU – General-purpose graphics processing units, обчислення загального призначення на графічних процесорах.

NVIDIA CUDA – Compute Unified Device Architecture, програмно-апаратна архітектура паралельних обчислень компанії NVIDIA.

ВСТУП

Актуальність дослідження обумовлена зростаючою потребою у високопродуктивних обчисленнях, що підтримують паралельне програмування, особливо в контексті швидкого розвитку технологій GPGPU. Останнім часом для організації та підтримки паралельного програмування розробникам доступні декілька технологій, зокрема GPGPU – можливість організації обчислень загального призначення на графічних процесорах (ядрах) звичайних дискретних відеокарт. Розробники, які використовують популярну IDE Microsoft Visual Studio та програмують мовами C та C++, мають доступ до пропрієтарної платформи CUDA від NVIDIA, яка працює з графічними процесорами серії GeForce. CUDA дозволяє розподіляти завдання між графічними ядрами для паралельного виконання, що є надзвичайно актуальним у контексті сучасних вимог до продуктивності та ефективності обчислень.

CUDA підтримує кілька мов програмування, таких як C, C++, Fortran, Python, що дозволяє програмістам використовувати зручний для них інструментарій. NVIDIA також надає ряд бібліотек для CUDA, таких як cuBLAS (бібліотека лінійної алгебри), cuFFT (бібліотека швидкого перетворення Фур'є), cuDNN (бібліотека глибокого навчання) і багато інших. Ці бібліотеки спрощують створення програм, орієнтованих на конкретні завдання.

CUDA підтримує багато різних моделей і серій графічних процесорів NVIDIA, що дозволяє розробникам створювати програми, які працюють на широкому спектрі обладнання. Організація коду для CUDA вимагає ретельного керування пам'яттю та правильного використання блоків та потоків. Також, деякі оптимізації можуть бути внесені для підвищення продуктивності, такі як використання спільної пам'яті та оптимізації доступу до глобальної пам'яті.

Метою виконання роботи є проектування, розробка, тестування та оцінка ефективності CUDA-реалізацій різних алгоритмів пошуку простих чисел.

Об'єктом роботи є методи та засоби організації процесу пошуку простих чисел з використанням ядр графічного процесора.

Предметом дослідження є техніки розробки та реалізації різних алгоритмів пошуку простих чисел з використанням програмно-апаратної платформи NVIDIA CUDA.

Для досягнення мети необхідно виконати наступні завдання:

1. виконати аналіз відомих алгоритмів пошуку простих чисел;

2. виконати огляд технології NVIDIA CUDA для програмної реалізації алгоритмів процесу пошуку простих чисел;

3. виконати опис структури та принципів роботи використаних алгоритмів;

4. виконати програмну реалізацію процесу пошуку простих чисел з використанням NVIDIA CUDA та з використанням CPU-обчислень;

6. провести збір, обробку, візуалізацію та аналіз отриманих даних;

7. визначити практичну значущість розробки.

Методи досліджень: методи системного аналізу для дослідження предметної області, методи розробки алгоритмів, методи програмної реалізації CUDA-застосунків, методи оцінювання продуктивності програмного забезпечення.

Практичною новизною роботи є встановлення факту, що використання CUDA-реалізацій окремих алгоритмів пошуку простих чисел значно зменшує час виконання та навантаження на центральний процесор.

Публікації та апробація кваліфікаційної роботи: публікація тези доповіді «Organization of prime number search process using NVIDIA CUDA» на VI Всеукраїнській науково-практичній інтернет-конференції «Сучасні технології в енергетиці, електромеханіці, системах управління та машинобудуванні» (м. Харків, 06-07 грудня 2023 р.).

1 ПОСТАНОВКА ЗАДАЧІ

1.1 Аналіз програмно-апаратних платформ обробки даних

Вирішення багатьох затребуваних завдань, таких як комп'ютерне моделювання, обробка відео, візуалізація, розпізнавання образів, обчислювальна біологія та хімія, сейсмічний аналіз, прогнозування часових рядів, фінансовий аналіз, варіаційне обчислення, чисельні методи, вимагає значних витрат процесорного часу на виконання обчислень. Використання математичних методів оптимізації вимагає величезної обчислювальної роботи, особливо великі труднощі виникають під час вирішення завдань оптимізації через велику кількість параметрів. Вирішити цю проблему та прискорити час обчислень можна шляхом паралельної обробки даних.

За останні десятиліття у всіх галузях науки значно зросла потреба у обробці величезної кількості даних. Для цього потрібні сильні обчислювальні системи. Через те, що стало неможливо збільшувати тактову частоту процесора, з'явилися багатоядерні архітектури, які вимагали навичок паралельного програмування. Через брак висококваліфікованих фахівців стало необхідним створення ефективних високорівневих технологій програмування. В результаті стався бурхливий розвиток суперкомп'ютерних технологій по всьому світу.

Проектуванням та розробкою високопродуктивних систем, заснованих на алгоритмах паралельної розробки, вже давно є підрозділи та лабораторії провідних IT-компаній Intel, Microsoft, Google, AMD, NVIDIA тощо. Паралельні обчислення привернули таку увагу завдяки стрімкому зростанню обсягів даних, які потребують обробки [3-4].

Графічні прискорювачі (GPU) стали використовуватися для неграфічних обчислень, що дозволило досягти прискорення роботи застосунків у десятки, а то й у сотні разів. Для написання програм, призначених для виконання на графічному процесорі, були розроблені спеціальні мови програмування: C-CUDA, Fortran-CUDA [2, 5, 27], OpenCL, OpenACC та інші.

Пікова продуктивність суперкомп'ютерів TOP500 зростає з великою швидкістю [17]. У першій трійці рейтингу TOP500 на листопад 2023 року присутні американські суперкомп'ютери: Frontier HPE Cray EX235a із продуктивністю 1679 петафлопс. Ця монструозна система включає 8,7 млн. обчислювальних ядр, а її споживана потужність сягає 22,7 МВт. За нею слідує Aurora HPE Cray EX з продуктивністю 1059 петафлопс. При кількості ядр 4,7 млн. споживає 24,7 МВт електроенергії. Третю позицію зайняв Eagle Microsoft NDv5 з продуктивністю 846 петафлопс. Саме він побудований на основі сучасних графічних прискорювачів NVIDIA H100, який надає забезпечує продуктивність, масштабованість та безпеку для будь-якого робочого навантаження. За допомогою системи перемикання NVIDIA NVLink можна підключити до 256 графічних процесорів H100 для прискорення робочих навантажень. Графічний процесор також містить спеціальний рушій для вирішення мовних моделей із трильйонами параметрів.

Сучасні центри обробки даних є ключовими для вирішення деяких найважливіших світових наукових та інженерних завдань. Графічні процесори центрів обробки даних NVIDIA кардинально змінюють економіку центру обробки даних, забезпечуючи продуктивність із значно меншою кількістю серверів, меншим енергоспоживанням і зменшенням накладних витрат на мережу, що призводить до загальної економії витрат у 5-10 разів [19].

На перший план у виготовленні суперкомп’ютерів у рамках ініціативи «IndiaAI Compute Capacity» намагається вийти Індія. Очікується, що в новому суперкомп'ютері буде використано що найменш 10000 графічних процесорів штучного інтелекту NVIDIA, яких було замовлено на 500 млн. доларів [18]. Ініціатива зосереджуватиметься на розширенні охоплення освіти випускників і аспірантів шляхом підвищення доступності програм штучного інтелекту.

Паралельне програмування – це метод програмування, в якому обчислення розподіляються між кількома обчислювальними ресурсами, такими як процесори, ядра процесорів або комп'ютери, з метою підвищення продуктивності та зменшення часу виконання завдань [6, 7]. Паралельне обчислення особливо

корисне на звичайних багатоядерних системах, де можна використовувати кілька ядр або логічних процесорів для вирішення однієї задачі одночасно.

Microsoft Visual Studio – одне з найбільш популярних інтегрованих середовищ розробки (IDE) у світі, особливо в корпоративному і комерційному секторі. Visual Studio підтримує мови програмування C++, C#, Visual Basic .NET, Python, F#, JavaScript. Для організації та підтримки паралельного програмування Visual Studio пропонує розробникам декілька підходів.

1. Використання потоків (Threads) дозволяє виконувати різні частини програми паралельно. Це основний спосіб паралельного програмування в багатьох мовах програмування, включаючи C#, C++, Java та Python.

2. Використання завдань (Tasks) – це вищий рівень абстракції для паралельного програмування і зазвичай є більш зручними для створення та керування паралельними завданнями у асинхронному режимі.

3. Стандарт для розподіленого обчислення MPI (Message Passing Interface), який дозволяє обмінюватися повідомленнями між процесами на різних вузлах великих кластерів або суперкомп'ютерів. Для підтримки MPI в Visual Studio потрібно встановити спеціальні розширення та бібліотеки.

4. OpenCL (Open Computing Language) – це відкритий стандарт для паралельного програмування обчислювальних пристроїв, таких як центральні процесори (CPU), графічні процесори (GPU), сопроцесори та інші.

5. OpenMP (Open Multi-Processing) – це стандарт для багатопоточного паралельного програмування в мовах програмування, що надає можливість створювати паралельні програми для багатоядерних систем.

6. GPGPU – можливість організації обчислень загального призначення на графічних процесорах. Одночасне використання декількох графічних чипів паралелізує природу обробки графіки. Звичайні дискретні відеокарти можна використовувати для організації паралельних обчислень у великій кількості областей, включаючи криптографію, машинне навчання і багато інших.

Усі ці фактори створюють унікальні можливості для розвитку технологій паралельних обчислень та дозволяють вирішувати складні завдання, які раніше

були недосяжними. Розробники повинні володіти навичками паралельного програмування та здатністю використовувати сучасні апаратні засоби для досягнення оптимальної продуктивності застосунків.

Вибір підходу до паралельного програмування залежить від конкретних потреб вашого проекту, доступних ресурсів і ваших знань у галузі паралельного програмування. Деякі підходи, зокрема MPI, OpenMP та OpenCL, обмежені використанням мови C або C++, тому далі не розглядаються.

Для платформі .NET поки що відсутня підтримка GPGPU, тому розробникам треба покладатися на сторонні рішення, зокрема бібліотеки Alea GPU, Atimesh Hybridizer, ILGPU, які можуть використовувати C# розробники.

Корпорація AMD пропонує власну GPGPU-технологію під назвою ATI Accelerated Parallel Processing – це набір вдосконалених апаратних і програмних технологій, які дозволяють графічним процесорам AMD, працюючи разом із CPU, прискорювати роботу застосунків, окрім графічних. Інтерфейс програмування здійснюється через OpenCL, який нажаль не підтримує мову C#. Але розробникам доступний ATI Stream SDK Developer, який підтримує мови програмування C++, Java та C# [9].

CUDA (Compute Unified Device Architecture) – це пропрієтарна система корпорації NVIDIA, що працює з графічними процесорами серії GeForce. Нажаль, API CUDA підтримує лише мови C та C++, але є можливість використовувати DLL-модулі CUDA в проєктах C# [8].

З CUDA програмісти можуть зосередитися на задачі розпаралелювання алгоритмів, а не витрачати час на їх реалізацію. Серійні частини програм запускаються на CPU, а паралельні порти вивантажуються на GPU. Таким чином, CUDA можна поступово застосовувати до існуючих програм. Процесор та графічний процесор розглядаються як окремі пристрої, які мають свої власні простори пам'яті. Ця конфігурація дозволяє одночасно обчислювати на CPU і GPU без конкуренції за ресурси пам'яті.

Графічні процесори з підтримкою CUDA мають сотні ядр, які можуть виконувати тисячі обчислювальних потоків. Ці ядра мають спільні ресурси,

включаючи файл регістрів та загальну пам'ять. Вбудована пам'ять, що спільно використовується, дозволяє паралельним завданням, що виконуються на цих ядрах, обмінюватися даними, не відправляючи їх по шині системної пам'яті.

1.2 Аналіз алгоритмів пошуку простих чисел

Натуральне число називається простим, якщо воно має лише два різні дільники: одиницю і саму себе. Завдання пошуку простих чисел не дає спокою математикам дуже давно. Довгий час прямого практичного застосування ця проблема не мала, але все змінилося з появою криптографії з відкритим ключем. Існують кілька способів пошуку простих чисел, що як представляють винятково академічний інтерес, так і застосовуваних сьогодні в різних галузях, зокрема у криптографії.

Решето Ератосфена – алгоритм, запропонований давньогрецьким математиком Ератосфеном [11]. Цей метод дозволяє знайти всі прості числа менше заданого числа *n*. Суть методу ось у чому. Візьмемо набір чисел від 2 до *n*. Викреслимо з набору (відсіємо) всі числа, що діляться на 2, крім 2. Перейдемо до наступного «не відсіяного» числа 3, знову викреслюємо все, що ділиться на 3. Переходимо до наступного числа 5 і так далі до тих пір, поки ми не дійдемо до *n*. Після виконання дій у початковому списку залишаться лише прості числа.

	2	3	4′	5	6″	7	8′	9′	10″
11	12″	13	14″	15″	16′	17	18″	19	20″
21″	22″	23	24″	25′	26″	27′	28″	29	30‴
31	32′	33″	34″	35″	36″	37	38″	39″	40″
41	42‴	43	44″	45″	46″	47	48″	49′	50″
51″	52″	53	54″	55″	56″	57″	58″	59	60‴
61	62″	63″	64′	65″	66‴	67	68″	69″	70‴
71	72″	73	74″	75″	76″	77″	78‴	79	80″
81′	82″	83	84‴	85″	86″	87″	88″	89	90‴
91″	92″	93″	94″	95″	96″	97	98″	99″	100‴

Рисунок 1.1 – Зображення процесу пошуку простих чисел решетом Ератосфена

Алгоритм можна дещо оптимізувати. Так як один із дільників складового числа n обов'язково $\leq \sqrt{n}$, алгоритм можна зупиняти, після викреслення чисел, що діляться на $\sqrt{n}$.

Складність алгоритму становить $O(n \log\log n)$, причому для зберігання інформації, які числа були викреслені, потрібно $O(n)$ пам'яті.

Існує низка оптимізації, що дозволяють знизити ці показники. Прийом під назвою «колісна факторізація» [12] полягає в тому, щоб включати до початкового списку лише взаємно прості числа з кількома першими простими числами (наприклад менше 30). Теоретично пропонується брати перші прості приблизно до $\sqrt{\log n}$. Це дозволяє знизити складність алгоритму в $\log\log n$ раз. Крім цього зменшення споживаної пам'яті використовується так зване сегментування. Початковий набір чисел ділиться на сегменти розміром $\leq \sqrt{n}$ і до кожного сегмента решето Ератосфена застосовується окремо.

Рисунок 1.2 – Зображення процесу пошуку простих чисел за допомогою *колісної факторізації*

Більш досконалий алгоритм відсіювання складених чисел отримав назву решето Аткіна [13]. На початковому етапі решето Аткіна є масивом A розміром n, заповнений нулями. Для визначення простих чисел перебираються усі $x, y < \sqrt{n}$. Для кожної такої пари обчислюється значення $4x^2 + y^2, 3x^2 + y^2, 3x^2 - y^2$ і значення елементів масиву $\mathrm{A}[4x^2 + y^2], \mathrm{A}[3x^2 + y^2], \mathrm{A}[3x^2 - y^2]$ збільшуються

на одиницю. Наприкінці роботи алгоритму індекси всіх елементів масиву, які мають непарні значення чи прості числа, чи квадрати простого числа. На останньому кроці алгоритму проводиться викреслення квадратів чисел, що залишилися в наборі.

Обчислювальна складність решета Аткіна та споживання пам'яті становлять $O(n)$. При використанні wheel factorization та сегментування оцінка складності алгоритму знижується до $O(n/\log\log n)$, а споживання пам'яті до $O(\sqrt{n})$.

При таких показниках складності, навіть оптимізоване решето Аткіна неможливо використовувати для пошуку по-справжньому великих простих чисел. Існують швидкі тести, що дозволяють перевірити, чи є задане число простим. На відміну від алгоритмів решета, такі тести не призначені для пошуку всіх простих чисел, вони лише здатні сказати з деякою ймовірністю, чи є кілька простих.

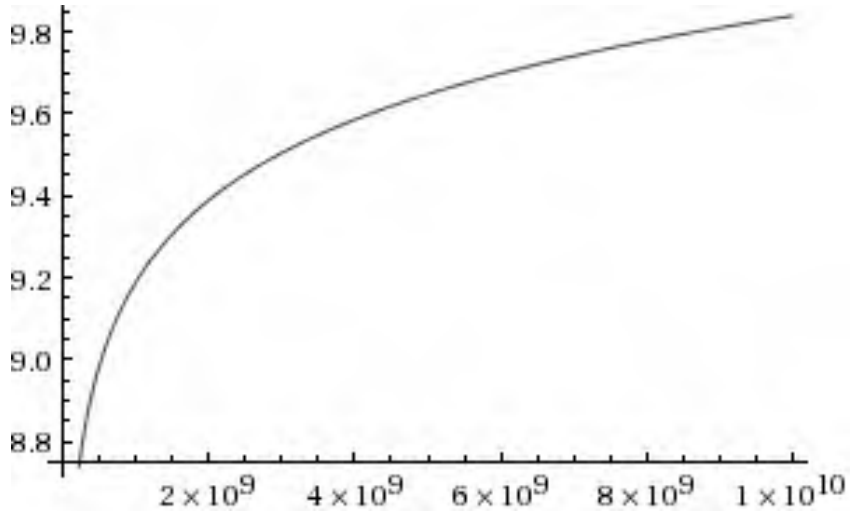


Рисунок 1.3 – Перевага решета Аткіна над решетом Ератосфена на числах від 2 до 10^9

Одним із таких методів перевірки є тест Люка-Лемера [14]. Це детермінований та безумовний тест простоти. Це означає, що тестування гарантує простоту числа. На жаль, тест призначений тільки для чисел особливого виду $2^p - 1$, де p – натуральне число. Такі числа називаються числами Мерсена. Завдяки простоті та детермінованості тесту, найбільші відомі прості числа – саме числа Мерсена. Найбільше відоме просте число на сьогодні $2^{82589933} - 1$ та складається з 24862048 цифр.

Для числа Мерсена довжиною p біт обчислювальна складність алгоритму становить $O(p^3)$. Простих чисел Мерсена відомо небагато, тому криптографії з відкритим ключем необхідний інший спосіб пошуку простих чисел. Одним із таких способів є тест простоти Ферма [15]. Він заснований на малій теоремі Ферма, яка свідчить, що якщо n – просте число, то для будь-якого a , яке не ділиться на n, виконується рівність $a^{n-1} \equiv 1 \ (mod\ n)$.

n	P_n	Digits	n	P_n	Digits
1	2	1	22	9941	2993
2	3	1	23	11213	3376
3	5	2	24	19937	6002
4	7	3	25	21701	6533
5	13	4	26	23209	6987
6	17	6	27	44497	13395
7	19	6	28	86243	25962
8	31	10	29	110503	33265
9	61	19	30	132049	39751
10	89	27	31	216091	65050
11	107	33	32	756839	227832
12	127	39	33	859433	258716
13	521	157	34	1257787	378632
14	607	183	35	1398269	420921
15	1279	386	36	2976221	895932
16	2203	664	37	3021377	909526
17	2281	687	38	6972593	2098960
18	3217	969	39	13466917	4053946
19	4253	1281	40	20996011	6320430
20	4423	1332	41	24036583	7235733
21	9689	2917	42	25964951	7816230
22	9941	2993	43	30402457	9152052
			44	32582657	9808358

Рисунок 1.4 – Перші 44 числа Мерсена (2^p-1)

Першим простим числом Мерсена, виявленим на комп'ютері за допомогою тесту Люка-Лемера, стало M_{521}, знайдене Рафаелем Робінсоном у 1952 р. на базі лампового комп'ютера SWAC. Число M_{44497} було обчислено у 1979 р. на суперкомп'ютері Cray-1. Число M_{219091} (30-е число Мерсена) було обчислено у 1985 р. на суперкомп'ютері Cray X-MP.

З відкриттям 34-го простого числа Мерсена $M_{1257787}$ у 1996 р. закінчилася епоха суперкомп'ютерів. Наступні 15 були знайдені добровольцями інтернет-проєкту GIMPS [20], в рамках якого проводиться варіант тесту Люка-Лемера на персональних комп'ютерах. Цей масштабний проект розподілених обчислень в даний час досягає рівня продуктивності, еквівалентного приблизно 300 терафлопс в секунду, причому задіяний час простою більш ніж 1,3 мільйонів комп'ютерів. Найбільших з відомих на даний момент простих чисел є 50-е число Мерсена $M_{77232917}$, а його довжина перевищує 23 мільйони символів.

Тест простоти Ферма – імовірнісний тест, який полягає у переборі кількох значень a якщо хоча б для одного з них виконується нерівність $a^{n-1} \neq 1\ (mod\ n)$, то число n – складове. В іншому випадку, n – ймовірно просте. Чим більше значень a використано у тесті, тим вища ймовірність того, що n – просте.

Складові (не прості) числа, які успішно проходять тест Ферма, називаються псевдопростими Ферма. Кількість псевдопростих Ферма нескінченна, тому тест Ферма – не надійний спосіб визначення простих чисел.

Більш надійних результатів можна досягти комбінуючи малу теорему Ферма і той факт, що для простого числа p немає інших коренів рівняння $x^2 \equiv 1\ (mod\ p)$, крім 1 та -1.

Тест Мілера-Рабіна [16] перебирає кілька значень a та перевіряє виконання наступних умов. Нехай p – просте число і $p - 1 = 2^s d$, тоді для будь-якого a справедлива хоча б одна з умов:

1. $a^d \equiv \pm 1\ (mod\ p)$;

2. Існує ціле число $r < s$ таке, що $a^{2^r d} \equiv -1\ (mod\ p)$.

По теоремі Ферма $a^{n-1} \equiv 1\ (mod\ n)$, тому, враховуючі, що $p - 1 = 2^s d$ з властивостей коріння рівняння $x^2 \equiv 1\ (mod\ p)$, ми знайдемо таке a, для якого одна з умов не виконується, отже p – складове число. Якщо одна з умов виконується, число a називають свідком простоти числа n за Мілером, а саме число n є ймовірно простим.

Чим більше свідків простоти знайдено, тим вища ймовірність того, що n – просте. Відповідно до теореми Рабіна ймовірність того, що випадково вибране

число a виявиться свідком простоти складового числа становить приблизно $1/4$. Отже, якщо перевірити k випадкових чисел a, то можливість прийняти складове число за просте $\approx (1/4)^k$. Складність роботи алгоритму $O(k \log^3 p)$, де k – кількість перевірок. Завдяки швидкості та високій точності тест Мілера-Рабіна широко використовується при пошуку простих чисел.

1.3 Профілювання навантажених програм у візуалізаторі MS Visualizer Concurrency

Здійснення відлагодження, оптимізації та перевірки результатів роботи програм обробки великих наборів даних потребує додаткові засоби, які б спрощували роботу програміста та дозволяли перевірити коректність роботи програми. Одним з таких засобів є програмне забезпечення є візуалізатор MS Concurrency Visualizer, який надає широкий спектр можливостей для профілювання та налагодження програм, дозволяє оптимізувати продуктивність програми, прослідкувати за поведінкою системи та її операцій [22, 25].

Навантажувальне тестування або тестування продуктивності – це автоматизоване тестування, яке імітує роботу бізнес-логіки програми на загальному ресурсі [23]. До результатів, одержуваних при тестуванні навантажень для подальшого їх аналізу, можна віднести, в першу чергу, споживання ресурсів центрального процесора. Ця метрика показує, який відсоток часу було витрачено процесором або процесорним ядром на обчислення вибраного процесу. У сучасних системах важливим фактором є здатність процесу працювати в декількох потоках, для того, щоб процесор міг проводити обчислення паралельно. Аналіз історії споживання ресурсів процесора може пояснювати вплив на загальну продуктивність системи потоків оброблюваних даних, конфігурації програми операційної системи, та інших факторів.

Профілювальник конкуренції або візуалізатор паралелізму під назвою Concurrency Visualizer, використовує події ETW (Event Tracing for Windows) для моніторингу продуктивності багатопотокових застосунків і надає візуальні звіти,

що спрощують пошук вузьких місць паралельних алгоритмів. Він встановлюється як компонент IDE Visual Studio та є безкоштовним.

Перегляди у Visualizer Concurrency надають графічні, табличні та текстові дані, які показують часові зв'язки між потоками програми та системою в цілому. Можна знайти недостатнє використання ядр, конкуренцію потоків, міграцію міжядерних потоків, затримки синхронізації, діяльність DirectX, області перекриття вводу-виводу. Представлення надають дані, з якими можна діяти, зв'язуючи їх графічний вихід із стеками викликів і вихідним кодом.

Перегляд «Threads» – це найбільш детальний і багатофункціональний режим для перегляду, які саме потоки виконують код під час сегмента виконання, і проаналізувати, чи потоки виконуються чи блокуються через синхронізацію, введення/виведення або з інших причин. Звіти про перегляд потоків також профілюють виконання дерева стека викликів і розблокування потоків.

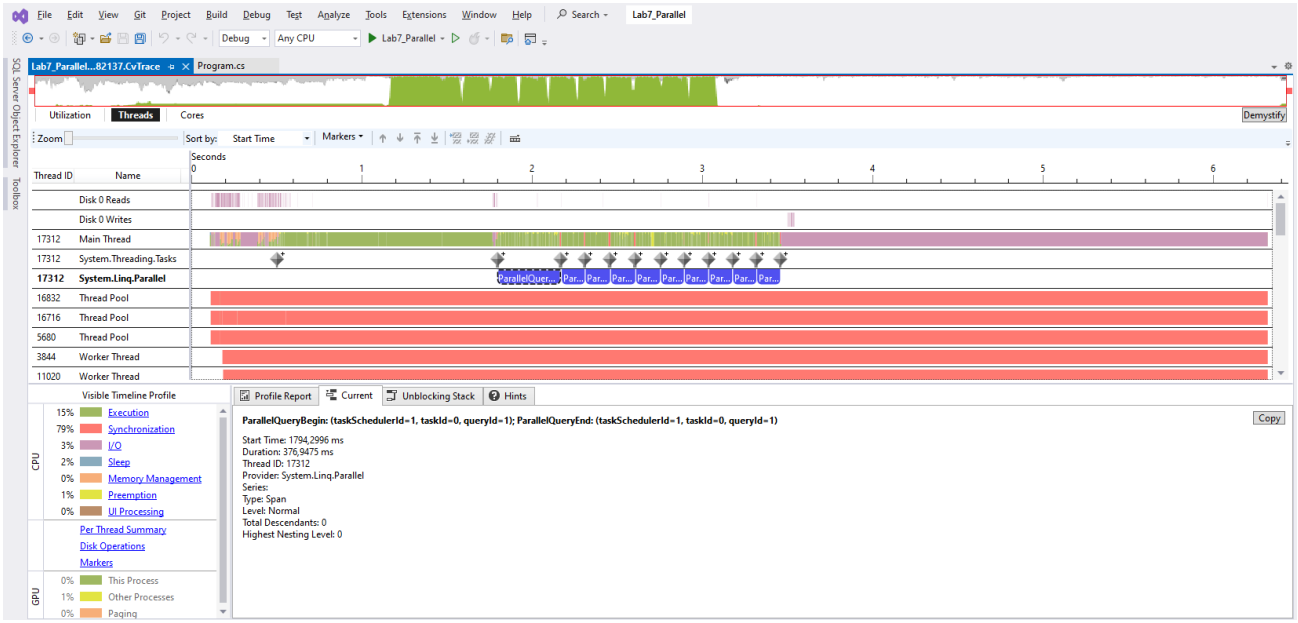


Рисунок 1.5 – Перегляд потоків у Visualizer Concurrency

1.4 Формулювання цілей та завдань дослідження

Аналіз сучасного стану проблеми пошуку простих чисел дозволив сформулювати наступні цілі та завдання дослідження:

1. виконати аналіз відомих алгоритмів пошуку простих чисел;

2. виконати огляд технології NVIDIA CUDA для програмної реалізації алгоритмів процесу пошуку простих чисел;

3. виконати опис структури та принципів роботи використаних алгоритмів;

4. виконати програмну реалізацію процесу пошуку простих чисел з використанням NVIDIA CUDA та з використанням CPU-обчислень;

6. провести збір, обробку, візуалізацію та аналіз отриманих даних;

7. визначити практичну значущість розробки.

1.5 Висновки за розділом

В результаті аналізу предметної галузі:

– встановлено, що сучасні програмно-апаратні платформи паралельної обробки даних широко використовують графічні процесори для прискорення складних обчислень;

– встановлено, що сучасні алгоритми пошуку простих чисел (решето Ератосфена, колісна факторізація, решето Аткіна, тести Люка-Лемера та Мілера-Рабіна) можуть бути реалізовані за допомогою доступних звичайному розробнику програмно-апаратних технологій;

– встановлено, що для оцінки продуктивності застосунків можна використати візуалізатор MS Visualizer Concurrency;

– виконано формулювання цілей та завдань подальшого дослідження.

2 ОПИС АЛГОРИТМІВ ТА ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

2.1 Огляд використовуваних технологій та інструментів

До складу платформи CUDA крім самої відеокарти входить драйвер CUDA, CUDA Toolkit та CUDA SDK. CUDA Toolkit включає бібліотеки, необхідні для роботи з платформою, і компілятор що транслює вихідний код програм в проміжний, а також додаткові бібліотеки. Драйвер CUDA включає компілятор, що здійснює перетворення проміжного асемблера в мікрокод, що виконується на GPU. CUDA SDK містить набір вихідних кодів простих програм, що ілюструють методи та можливості роботи з платформою CUDA.

Поява багатоядерних і багатопроцесорних графічних процесорів означає, що основні процесори тепер є паралельними системами. Завдання полягає в тому, щоб розробити прикладне програмне забезпечення, яке прозоро масштабує свій паралелізм, щоб використовувати все більше процесорних ядр. Модель паралельного програмування CUDA призначена для вирішення цього завдання для програмістів, знайомих з мовою програмування C++.

Модель CUDA надає розробнику три ключові абстракції – ієрархія груп потоків, спогадів та бар'єрної синхронізації, які надаються як мінімальний набір мовних розширень. Ці абстракції забезпечують дрібнозернистий паралелізм даних та паралелізм потоків, вкладені у грубий паралелізм даних та паралелізм завдань. Вони направляють програміста на те, щоб розділити проблему на грубі підзадачі, які можуть вирішуватися незалежно паралельно блоками потоків, а кожне підзавдання – у дрібніші частини, які можуть спільно вирішуватися спільно з усіма потоками всередині блоку.

З програмної точки зору CUDA є дещо інше, як розширення мов програмування C/C++, але з деякими інструкціями для їх відхилення від host-коду, а також інструкції для розрізнення типів пам'яті даних, які існують на графічному процесорі. Такі функції можуть мати параметри, і їх можна викликати за допомогою синтаксису, який дуже схожий на звичайний виклик функції C, але

трохи розширений для можливості вказати матрицю потоків GPU, які повинні виконувати функцію, що викликається. Протягом свого життєвого циклу host-процес може надсилати багато паралельних завдань графічного процесора.

Вихідні файли для CUDA складаються з комбінації звичайного host-коду C++ та функцій пристрою GPU. Трасування CUDA-компіляції відокремлює функції пристрою від головного коду, компілює функції пристрою з використанням власних компіляторів NVIDIA, компілює host-код з використанням компілятора C++.

Ключові особливості програмної сторони CUDA включають:

– специфікатори функцій, які показують, як і звідки виконуватимуться функції;

– специфікатори змінних, які служать для вказівки типу пам'яті GPU, що використовується;

– специфікатори запуску ядра GPU;

– вбудовані змінні для ідентифікації тредів та блоків;

– додаткові типи змінних.

Специфікатори визначають виклик функції, а всього їх 3:

1) __global__, виконання програми відбувається на GPU, а виклик здійснюється з CPU;

2) __device__, виконання та виклик з GPU;

3) __host__, виконання та виклик з CPU.

Фізично пам'ять GPU поділяється на DRAM і мультипроцесорну, розміщену безпосередньо на самому графічному процесорі.

У CUDA є всього шість типів пам'яті: реєстри; локальна пам'ять; пам'ять, що розділяється; світова пам'ять; текстурна пам'ять; константна пам'ять. Найбільш простим видом пам'яті є реєстрова пам'ять. Кожен потоковий мультипроцесор містить 8192 або 16384 32-бітових регістра.

Кожен блок потоків може бути запланований на будь-якому з доступних мультипроцесорів в GPU, у будь-якому порядку: паралельно чи послідовно. Тому вже скомпільована програма на основі CUDA може виконуватися на будь-якій

кількості мультипроцесорів (рис. 2.1). Одним обмеженням є те, що програма має отримати фізичну кількість мультипроцесорів. Ця перевага дає можливість архітектурі GPU зайняти широкий спектр: від високопродуктивних до недорогих графічних процесорів.

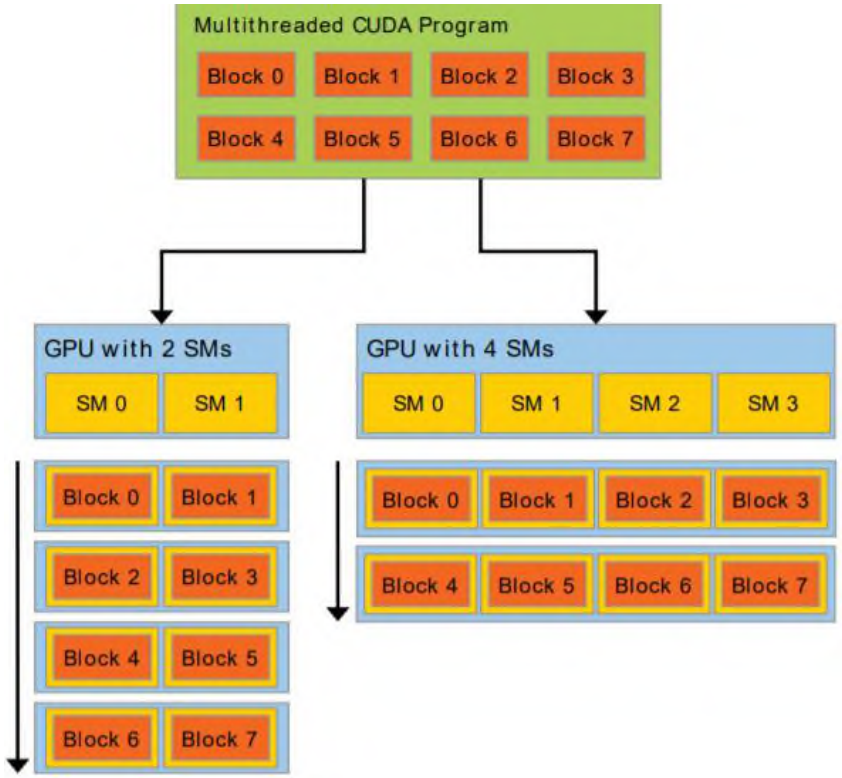


Рисунок 2.1 – Схема роботи CUDA-програми на GPU з різною кількістю мультипроцесорів

Локальна пам'ять – це область глобальної пам'яті, виділена компілятором для зберігання локальних значень потоків. Локальна пам'ять використовується для зберігання локальних даних потоків у разі нестачі регістрів або оголошення локальних масивів усередині ядра.

Розділена (shared) пам'ять розташована безпосередньо в мультипроцесорі, але вона виділяється на рівні блоків –кожен блок отримує в своє розпорядження одну і ту ж кількість пам'яті, що розділяється. Всього кожен мультипроцесор містить 16 Кбайт пам'яті, що розділяється, і від того, скільки розділяється пам'яті потрібно блоку, залежить кількість блоків, яке може бути запущено на одному мультипроцесорі.

Глобальна пам'ять – це звичайна DRAM-пам'ять, яка виділяється з

допомогою спеціальних функцій на CPU. Оскільки глобальна пам'ять розташована на центральному процесорі, а не на графічному, то вона має високу латентність (від 400 до 600 тактів). Вона є основною пам'яттю у CUDA, вона повільно звертається за випадковою адресою, відсутній кеш і кожного разу дані треба зчитувати заново.

2.2 Розробка алгоритмів пошуку простих чисел

Для організації процесу пошуку простих чисел решетом Ератосфена засобами NVIDIA CUDA необхідно виконати наступні дії.

1. Розбиття діапазону чисел, для якого необхідно знайти прості числа, на менші інтервали. Кожен інтервал буде оброблюватися окремим блоком CUDA.

2. Обираємо алгоритм для решета Ератосфена.

3. Виконаємо розподіл обчислень між потоками CUDA. Кожен потік буде обробляти свій власний піддіапазон чисел.

4. Для організації паралельних обчислень необхідно виконати розподіл інтервалів між блоками і потоками.

5. Для уникнення колізій та правильного об'єднання результатів використовуємо механізми синхронізації CUDA, такі як «syncthreads».

6. Організовуємо процес збереження та обробки результатів обчислень на головному процесорі.

7. Використовуємо оптимізацію CUDA, такі як shared memory для обміну даними між thread у межах блоку, що може покращити швидкість обчислень.

8. Після отримання результатів з різними діапазонами чисел та їх верифікації за допомогою довідників, необхідно виконати аналіз продуктивності обчислень за допомогою візуалізатору паралелізму Concurrency Visualizer.

На рис. 2.2 показано розроблений алгоритм використання решета Ератосфена для знаходження простих чисел в заданому діапазоні засобами CUDA для паралельного обчислення на графічних пристроях.

Програма використовує CUDA-ядро для знаходження простих чисел за допомогою решета Ератосфена. Спочатку ініціалізуємо масив чисел, вважаючи їх

всі простими. Потім використовується CUDA-ядро для паралельного викреслення кратних чисел із масиву. Після завершення обчислень результат копіюється назад на host-частину. На виході програма виводить діапазон пошуку, час виконання, кількість та найбільше просте число, знайдені за допомогою решета Ератосфена.

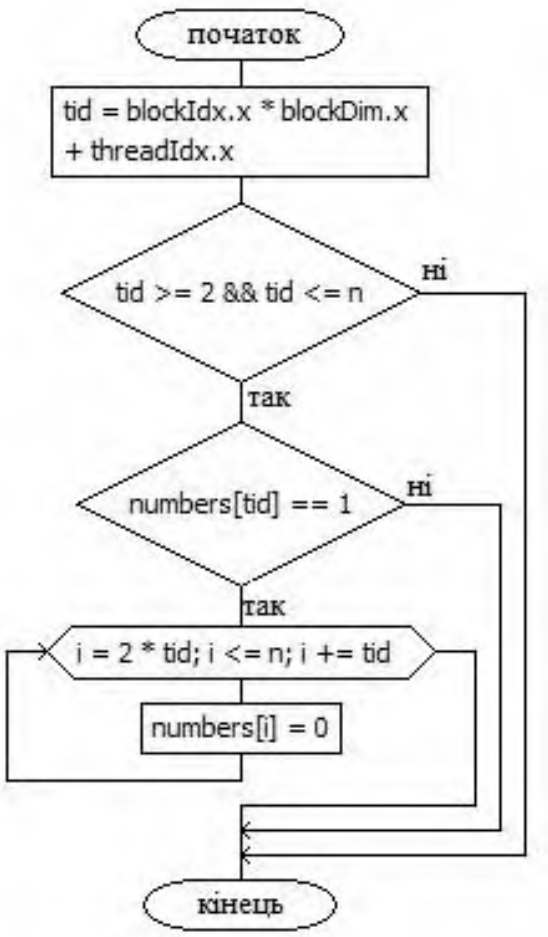


Рисунок 2.2 – Блок-схема алгоритму знаходження простих чисел решетом Ератосфена для CUDA-ядра

Враховуючи те, що Windows-застосунки будуть оперувати даними великого об’єму, необхідно розробити модуль для введення-виведення даних. Прийнято рішення реалізувати процес вводу-виводу даних з файлу у форматі JSON (JavaScript Object Notation) – це формат обміну даними, який базується мовою JavaScript. Він використовується для представлення структурованих даних у вигляді пар ключ-значення, де дані можуть бути масивами, об'єктами, числами, рядками, логічними значеннями або null. На рис. 2.3 показано блок-схему алгоритму генерації файлу зі знайденими простими числами у форматі Json.

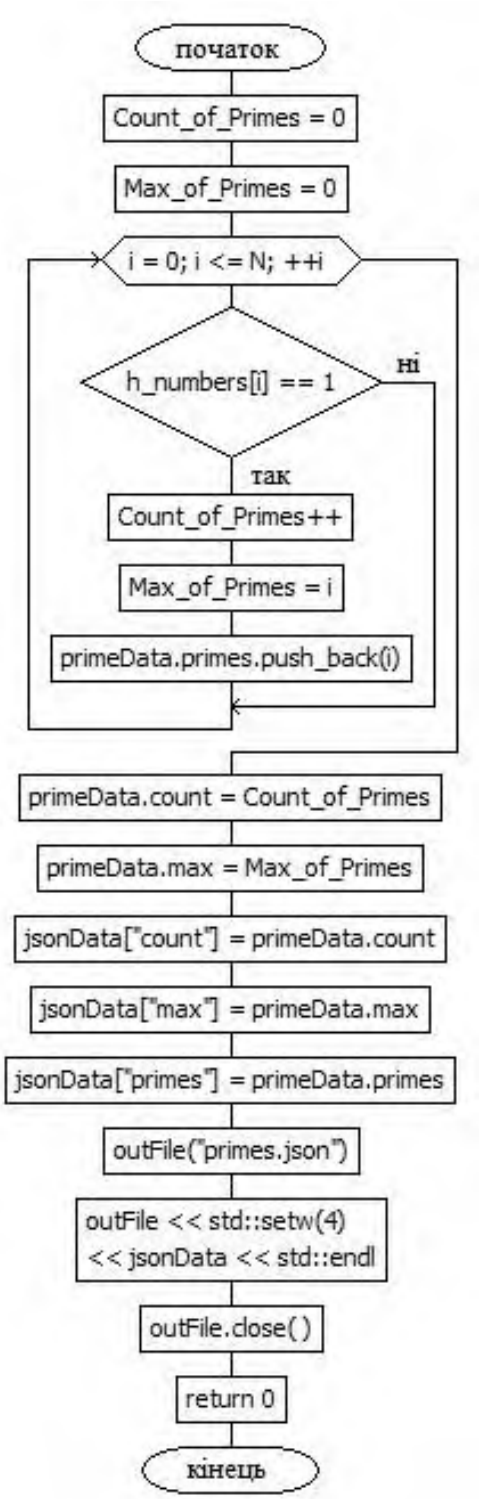


Рисунок 2.3 – Блок-схема процесу зберігання простих чисел у JSON-файлі

Виконано розробку алгоритму знаходження простих чисел у заданому діапазоні колісною факторизацією засобами CUDA для паралельного обчислення на графічних пристроях. Основний алгоритм можна розбити на кілька етапів (рис. 2.4).

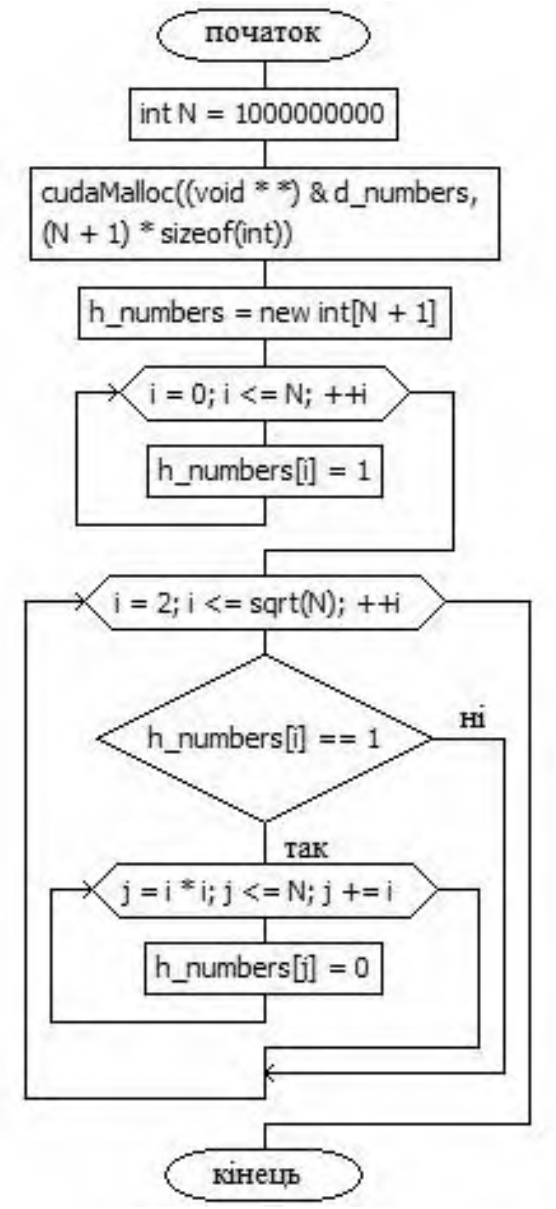


Рисунок 2.4 – Блок-схема алгоритму пошуку простих чисел за допомогою колісної факторизації для CUDA-ядра

1. Ініціалізація: Спочатку відбувається резервування пам'яті для масиву чисел на пристрої CUDA та у host-частину. Після цього масив ініціалізується з використанням колісної факторизації, що дозволяє визначати, чи число є простим.

2. Передача даних на пристрій: Заповнений масив чисел копіюється з hosta на пристрій CUDA.

3. CUDA-ядро виконує алгоритм решета Ератосфена. Кожен потік обробляє певний діапазон чисел. Якщо число є простим, воно залишається без змін, а якщо не є – всі його кратні числа відмічаються як непрості.

4. Результати обчислень копіюються з GPU у host-частину.

5. Підрахунок та вивід результатів: на основі отриманих даних підраховуються кількість знайдених простих чисел та найбільше з них. Ці дані виводяться на екран.

6. Очищення пам'яті: Після завершення роботи програми пам'ять, яка була зарезервовано, звільняється.

Виконано розробку алгоритму знаходження простих чисел у заданому діапазоні за допомогою решета Аткіна засобами CUDA для паралельного обчислення на графічних пристроях. Основний алгоритм можна розбити на кілька етапів (рис. 2.5). Основний алгоритм має наступні кроки.

1. Ініціалізація: Резервується пам'ять для масиву чисел на пристрої CUDA (cudaMalloc) та на hosti (new). Масив чисел ініціалізується нулями, що означає, що спочатку всі числа вважаються непростими.

2. CUDA-ядро виконує алгоритм решета Аткіна. Кожен потік обробляє певний діапазон чисел. Для кожного числа, що потрапляє в діапазон, обчислюються значення згідно з алгоритмом Аткіна, і якщо вони відповідають умовам, відповідні елементи масиву чисел інвертуються.

3. Передача даних на host: Результати обчислень (зокрема, знайдені прості числа) копіюються з пристрою на host за допомогою cudaMemcpy.

4. На основі отриманих даних підраховуються кількість знайдених простих чисел та найбільше з них. Ці дані виводяться на екран.

5. Після завершення роботи програми пам'ять звільняється.

Виконано розробку алгоритму знаходження простих чисел у заданому діапазоні за допомогою тесту Міллера-Рабіна засобами CUDA для паралельного обчислення на GPU. Основний алгоритм можна розбити на кілька етапів (рис. 2.6).

1. Функція millerRabinTest виконує тест Міллера-Рабіна для перевірки, чи є число простим. Вона проводить кілька ітерацій, кожна з яких генерує випадкове число a у діапазоні від 2 до n - 2 і обчислює (a^d) % n, де d є цілим числом таким, що n - 1 = 2^s * d. Якщо результат не задовольняє деякі умови, то число вважається складеним.

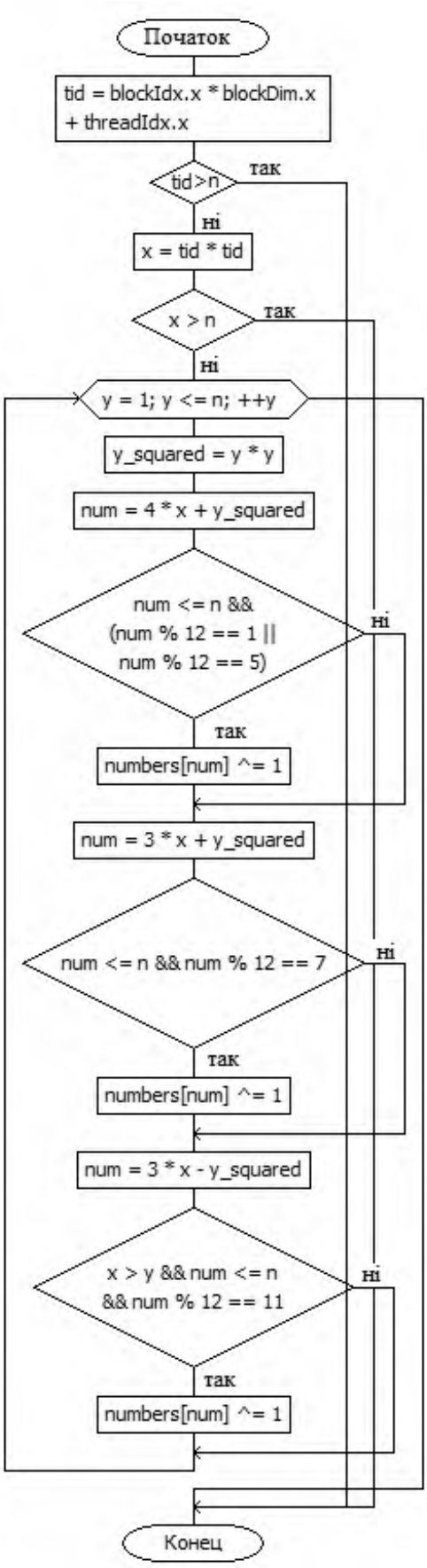


Рисунок 2.5 – Блок-схема алгоритму пошуку простих чисел решетом Аткіна для CUDA-ядра

2. CUDA-ядро виконує тест Міллера-Рабіна для кожного числа у заданому діапазоні. Кожен потік обробляє певне число з діапазону. Використовується генератор випадкових чисел. Результати тесту записуються у масив primes.

3. В головній функції задаються параметри, виконується виділення пам'яті для масиву чисел на пристрої CUDA, запуск CUDA-ядра для пошуку простих чисел, копіювання результатів назад у host-частину, обчислення часу виконання програми та виведення результатів на екран.

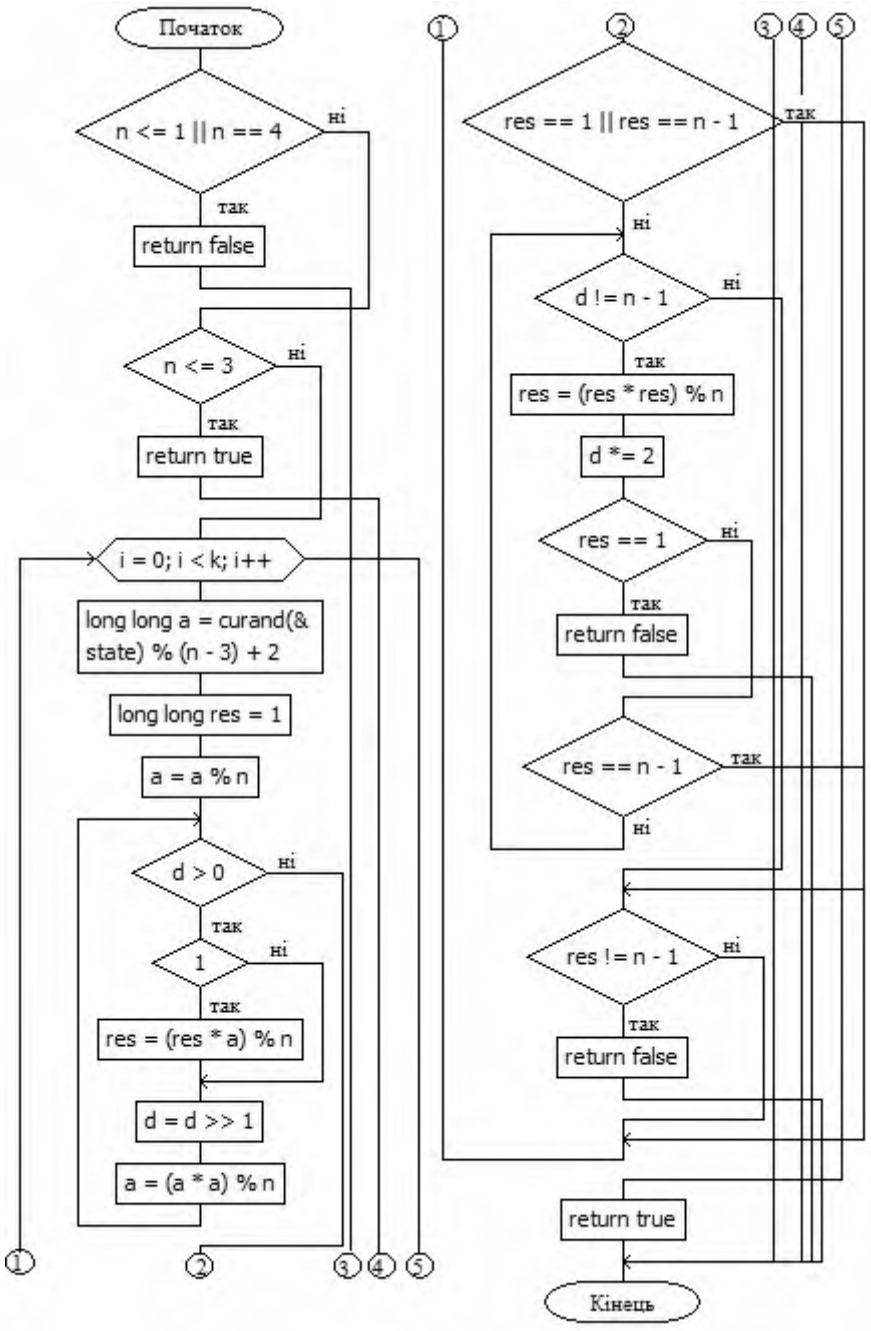


Рисунок 2.6 – Блок-схема алгоритму пошуку простих чисел за допомогою тесту Мілера-Рабіна для CUDA-ядра

Виконано розробку алгоритму знаходження простих чисел Мерсена у заданому діапазоні за допомогою тесту Люка-Лемера засобами CUDA для паралельного обчислення на графічних пристроях. Основний алгоритм можна розбити на кілька етапів (рис. 2.7).

1. Функція lucasLehmerTest виконує тест Люка-Лемера для кожного числа у заданому діапазоні. Для кожного числа p від 3 до n, де n – верхня межа, визначається число Мерсена за формулою 2^p–1. Далі виконується рекурентна послідовність Люка-Лемера, де кожен елемент s_i = s_{i-1}^2–2, де s_0= 4. Якщо останній елемент послідовності s_{p-2} = 0, то число 2^p - 1 є простим.

2. CUDA-ядро для виконання тесту Люка-Лемера запускається на пристрої CUDA з параметрами блоку та сітки. Кожен потік обробляє своє число. Визначається число Мерсена та виконується тест Люка-Лемера.

3. В головній функції програми задаються параметри, виділяється пам'ять для масиву чисел на пристрої CUDA, запускається CUDA-ядро для виконання тесту Люка-Лемера, копіюються результати на host-частину, обчислюється час виконання програми та виводяться результати на екран.

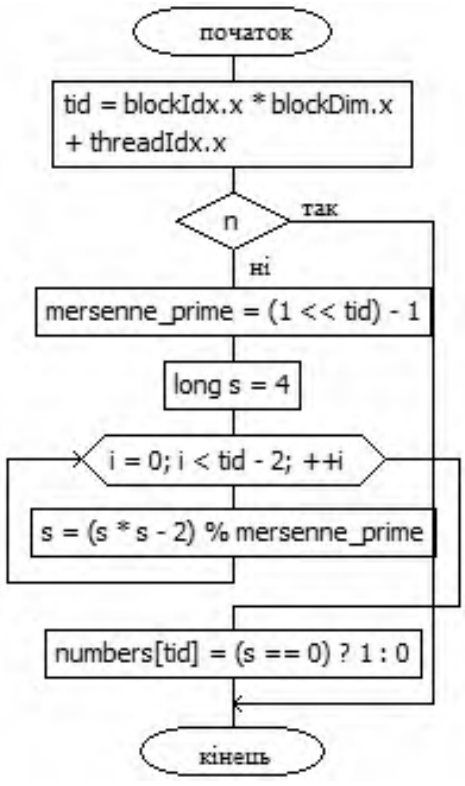


Рисунок 2.7 – Блок-схема алгоритму пошуку чисел Мерсена за допомогою тесту Люка-Лемера для CUDA-ядра

2.3 Опис ключових моментів програмної реалізації

Основний код програми міститься у файлах cernel.cu у кожному з CUDA-застосунків і представлений у вигляді набору функцій, що виконуються на графічному пристрої.

Далі наведено приклад коду CUDA-ядра ініціалізації генератора випадкових чисел для пошуку множників числа за тестом Мілера-Рабіна:

```
__device__ bool millerRabinTest(unsigned long long n, int k, curandState& state,
unsigned long long d) {
    // Проводимо k ітерацій тесту Міллера-Рабіна
    for (int i = 0; i < k; i++) {
        // Вибираємо випадкове ціле число у діапазоні [2, n-2]
        unsigned long long a = curand(&state) % (n - 3) + 2;

        // Знаходимо a^d % n
        unsigned long long res = 1; // Ініціалізуємо результат
    }

    // Якщо всі тести пройдені, n має високий шанс бути простим
    return true;
}
```

Приклад коду CUDA-ядра генерації глобального ідентифікатору потоку даних, надісланих на графічний пристрій:

```
__global__ void sieveOfEratosthenes(int* numbers, int n) {
    // Отримуємо глобальний ідентифікатор потоку
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    // Виключаємо 0 і 1 з решета
    if (tid >= 2 && tid <= n) {
        // Якщо число не викреслене, то викреслюємо його кратні
        if (numbers[tid] == 1) {
            for (int i = 2 * tid; i <= n; i += tid) {
                numbers[i] = 0;
            }
        }
    }
}
```

Приклад коду CUDA-ядра для знаходження простих чисел за допомогою решета Аткіна:

```
__global__ void sieveOfAtkin(int* numbers, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid <= 2 || tid > n) return;

    int x = tid * tid;
    if (x > n) return;
    for (int y = 1; y <= n; ++y) {
        int y_squared = y * y;
        int num = 4 * x + y_squared;
        if (num <= n && (num % 12 == 1 || num % 12 == 5))
            numbers[num] ^= 1;
```

```
        num = 3 * x + y_squared;

        if (num <= n && num % 12 == 7)
            numbers[num] ^= 1;

        num = 3 * x - y_squared;

        if (x > y && num <= n && num % 12 == 11)
            numbers[num] ^= 1;
    }
}
```

Приклад коду host-частини для виділення необхідної кількості пам'яті на пристрої CUDA-ядра:

```
    const int N = 1000000;  // Верхня межа пошуку простих чисел
    clock_t begin = clock(); // Початок таймера
    int* d_numbers;
    cudaMalloc((void**)&d_numbers, (N + 1) * sizeof(int));
    int* h_numbers = new int[N + 1];

    // Ініціалізуємо масив чисел за допомогою решета Аткіна
    for (int i = 0; i <= N; ++i) {
        h_numbers[i] = 0;
    }

    int blockSize = BLOCK_SIZE;
    int gridSize = (N + blockSize - 1) / blockSize;
    sieveOfAtkin << <gridSize, blockSize >> > (d_numbers, N);

    cudaMemcpy(h_numbers, d_numbers, (N + 1) * sizeof(int),
               cudaMemcpyDeviceToHost);
```

Функція синхронізації виконання ядр на GPU-пристрої:

```
device void SyncWholeGPU()
{
    unsigned int old;
    threadfence();
    {
        old = atomicInc((unsigned int *) & count, gridDim.x - 1);
        threadfence();
        if (old! = (gridDim.x - 1)) while (count! = 0) ;
    }
    syncthreads();
}
```

Основні етапи виконання CUDA-програми:

1. Host-частина програми виділяє необхідну кількість пам'яті на графічному пристрої:

```
    cudaMalloc((void**)&d_numbers, (N + 1) * sizeof(int));
```

2. Host-частина копіює дані зі своєї пам'яті в пам'ять графічному пристрою:

```
    cudaMemcpy(h_numbers, d_numbers, (N) * sizeof(int), cudaMemcpyDeviceToHost);
```

3. Основним будівельним блоком програми є потік даних thread. Всі потоки групуються в warps, які групуються в блоки, які в кінцевому підсумку містяться в

grid. Host-частина стартує виконання певних ядр на графічному пристрої, з вказівкою параметрів запуску ядр:

```
dim3 blockSize; dim3 gridSize; int threadNum;
// blockSize – розмір блоку, що задається
// gridSize – Розмір сітки
gridSize = dim3(instances, 1, 1);
// instances – кількість екземплярів обчислень функції, що запускаються
// Завдання розмірності блоку
// blockSize =dim3(threadNum, 1, 1);
// максимально можлива кількість потоків на пристрої
threadNum = 1024;
```

4. Графічний пристрій виконує код CUDA-ядра:

```
device void Ordinary_Search << <gridSize, blockSize> >>
    ( current_square, Fn, func_type, credits);
devicevoid Hypersquare_Search << <gridSize, blockSize> >>
    ( current_square, Fn, func_type, credits);
```

Host-частина програми копіює обчислені результати з пам'яті графічного пристрою у пам'ять. Для максимальної ефективності використання GPU необхідно щоб співвідношення часу, витраченого на роботу ядр, до часу, витраченого на виділення пам'яті та переміщення даних, було якнайбільше.

2.4 Висновки за розділом

За результатами аналізу предметної області виконана розробка алгоритмів пошуку простих чисел та вибрано засоби розробки відповідного програмного забезпечення, за допомогою яких вдасться реалізувати мету дослідження.

Під час проектування отримано наступні результати:

– розроблено блок-схеми популярних алгоритмів знаходження простих чисел в заданому діапазоні засобами CUDA для паралельного обчислення на графічних пристроях, а саме: решето Ератосфена, колісна факторизація, решето Аткіна, а також алгоритмів знаходження простих чисел Мерсена за допомогою тестів Люка-Лемера та Міллера-Рабіна;

– розглянута модель паралельного програмування NVIDIA CUDA;

– для програмної реалізації алгоритмів використано програмно-апаратну платформу NVIDIA CUDA, IDE Visual Studio та мову програмування C++.

3 РЕЗУЛЬТАТИ ЕКСПЕРИМЕНТІВ ТА АНАЛІЗ ОТРИМАНИХ ДАНИХ

3.1 Тестування програмної CPU- та CUDA-реалізації алгоритмів пошуку простих чисел

З метою перевірки працездатності розробленого програмного забезпечення та оцінки приросту швидкодії здійснено низку тестових запусків CPU-реалізацій та CUDA-реалізацій алгоритмів пошуку простих чисел: решето Ератосфена, колісної факторизації, решето Аткіна, тест Мілера-Рабіна Мілера-Рабіна. Тестування проводилося на різних діапазонах чисел: 0÷1 000 000, 0÷10 000 000, 0÷100 000 000, 0÷1000 000 000.

Тестування проводилося в двох режимах: виконання на CPU, виконання на NVIDIA CUDA. Такий підхід дозволяє наочно побачити різницю в часі виконання між вибраними методами і оцінити отриманий приріст швидкодії. З метою зменшення похибки експеримент був проведений кілька разів та вибрано найбільш сталі показники. Для тестування вибрана обчислювальна система, програмно-апаратні характеристики якої показано на рис. 3.1.

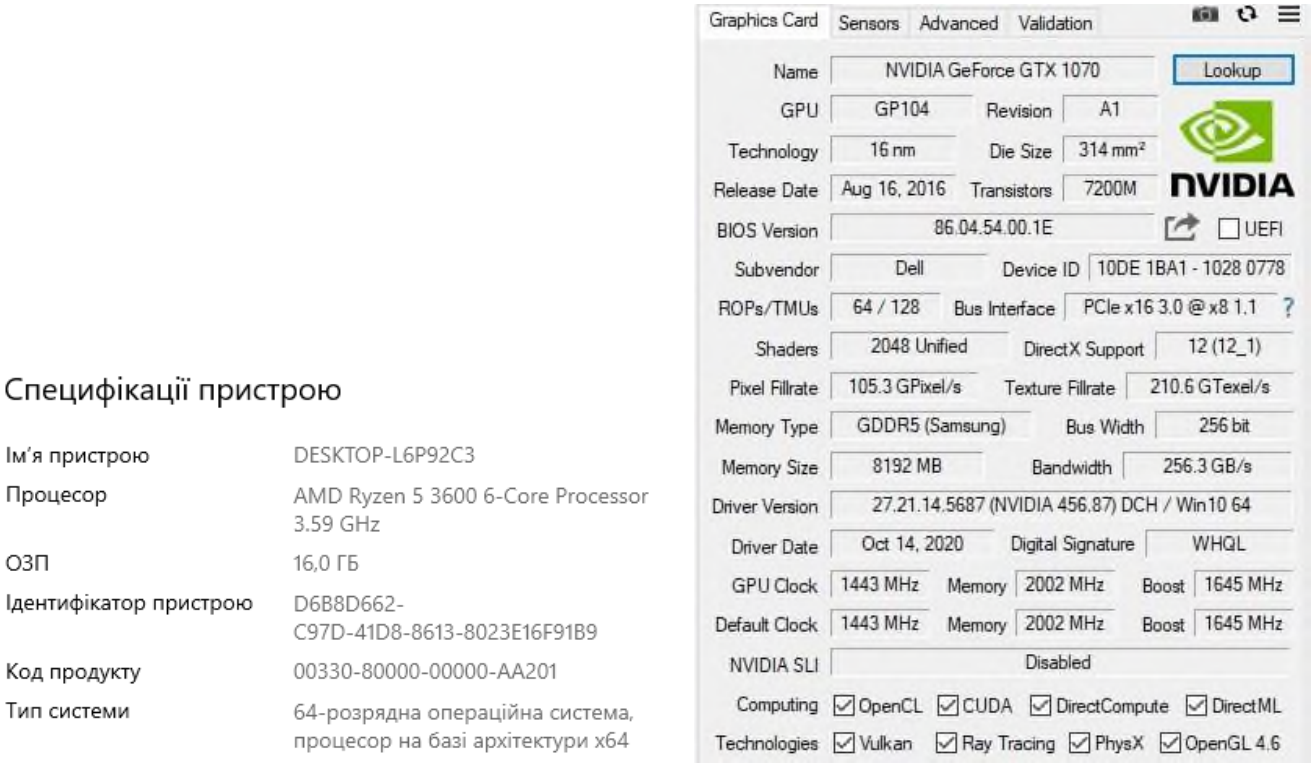


Рисунок 3.1 – Характеристики обчислювальної системи

Виконано тестування програмної CPU-реалізації та CUDA-реалізації для знаходження простих чисел у діапазоні за допомогою решета Ератосфена. Для порівняння виконана багатопотокова паралельна реалізація мовою C# [24]. Результати роботи на діапазоні $0 \div 1\,000\,000\,000$ показано на рис. 3.2-3.4.

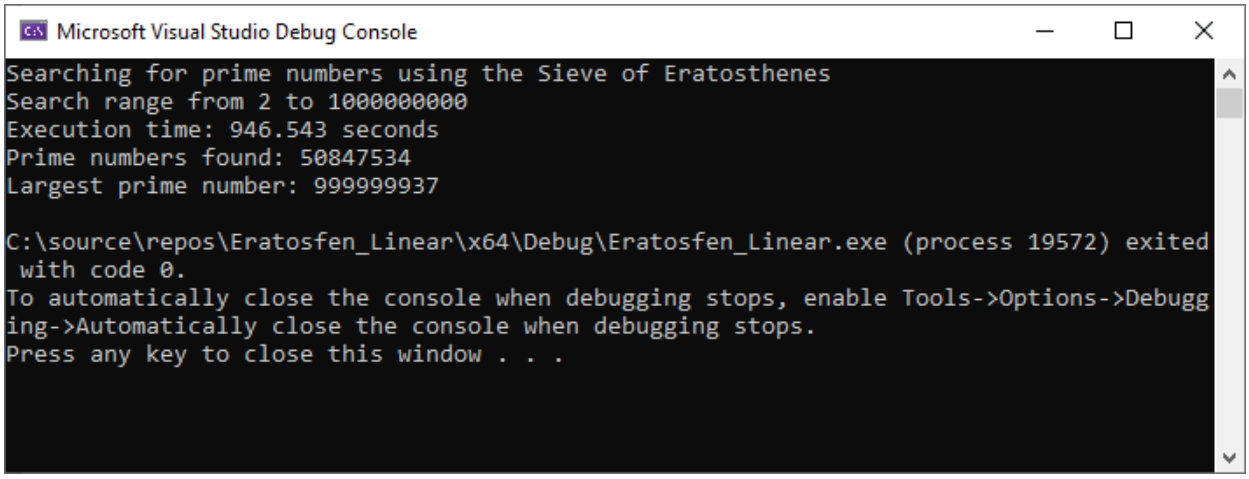


Рисунок 3.2 – Результати пошуку простих чисел у діапазоні 0÷1 000 000 000 решетом Ератосфена з CPU-реалізацією (лістинг у дод. Б Б)

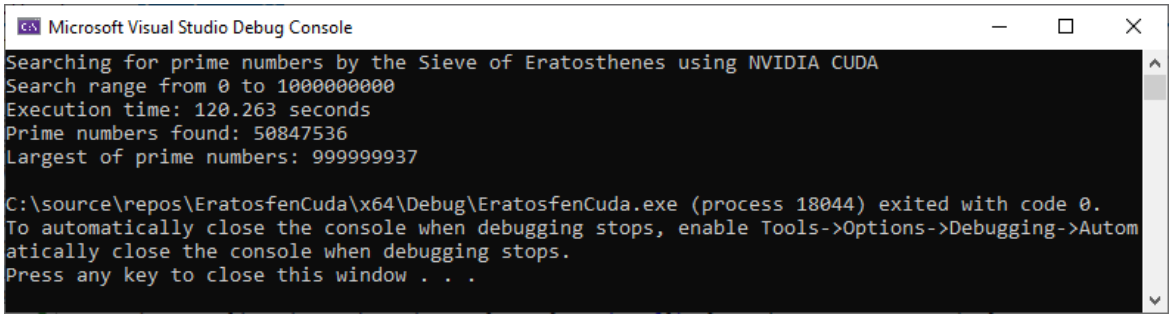


Рисунок 3.3 – Результати пошуку простих чисел у діапазоні 0÷1 000 000 000 решетом Ератосфена з CUDA-реалізацією (лістинг у дод. Б)

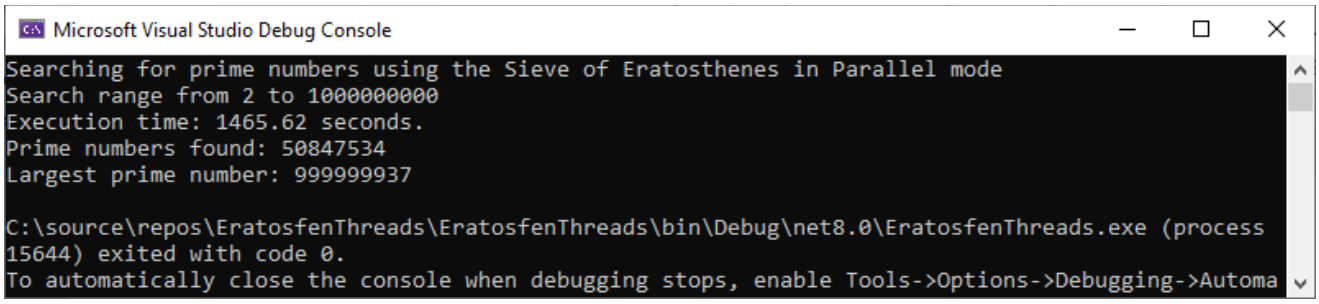


Рисунок 3.4 – Результати пошуку простих чисел у діапазоні 0÷1 000 000 000 решетом Ератосфена з багатопотоковою паралельною реалізацією

Виконано тестування програмної CPU-реалізації та CUDA-реалізації для знаходження простих чисел у заданому діапазоні за допомогою колісної факторизації. Процес розглядається як обертання віртуального "колеса", яке містить числа з певними властивостями, наприклад, лише ті, що не кратні 2, 3 та 5. Кожен множник (просте число) використовується як "шип" на цьому колесі, який вказує, які числа насправді є простими. Результати роботи застосунків на діапазоні $0 \div 1\,000\,000\,000$ показано на рис. 3.5-3.6.

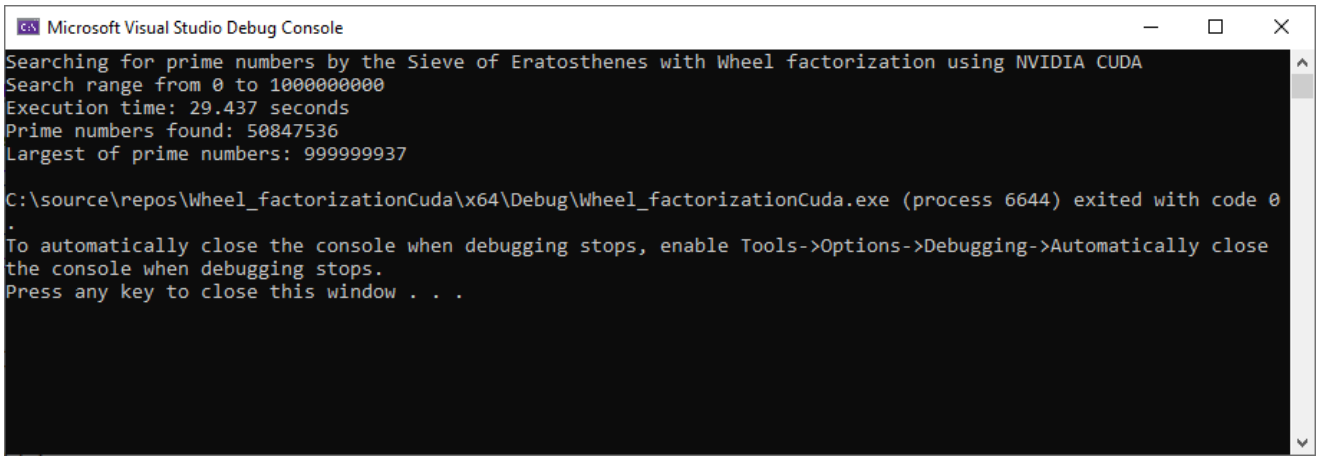


Рисунок 3.5 – Результати пошуку простих чисел у діапазоні 0÷1 000 000 000 з колісною факторизацією та CUDA-реалізацією (лістинг у дод. Б)

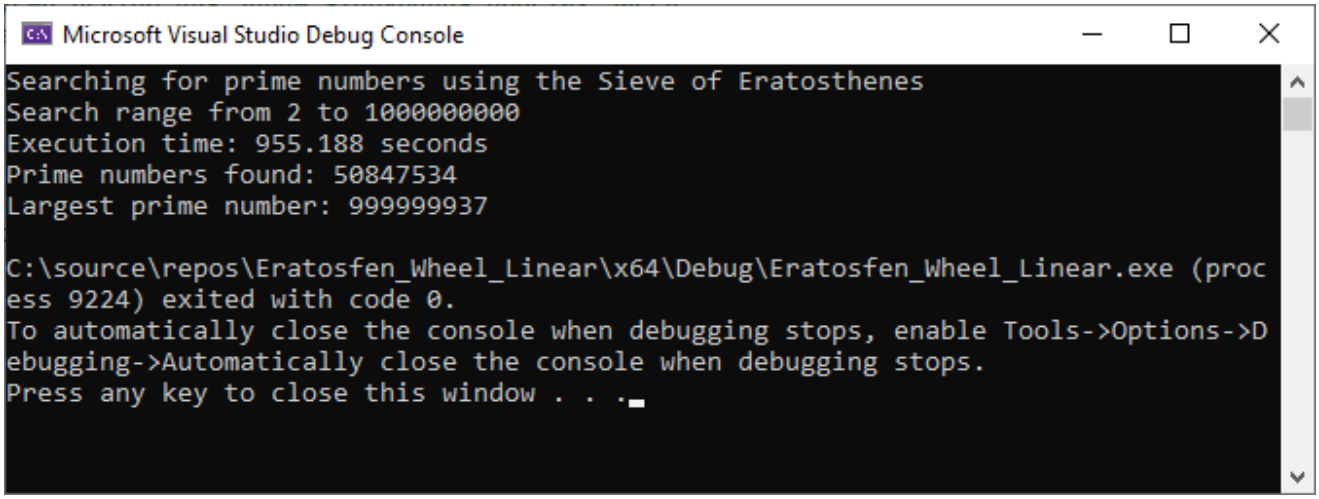


Рисунок 3.6 – Результати пошуку простих чисел у діапазоні 0÷1 000 000 000 з колісною факторизацією та CPU-реалізацією (лістинг у дод. Б)

Виконано тестування програмної CPU-реалізації та CUDA-реалізації для знаходження простих чисел з використанням решета Аткіна. Цей алгоритм

дозволяє більш ефективно знаходити прості числа, ніж решето Ератосфена, за рахунок використання принципів квадратичних форм та реалізації більш складних перевірок на простоту чисел. Результати роботи на діапазоні 0÷1000000 показано на рис. 3.7-3.8.

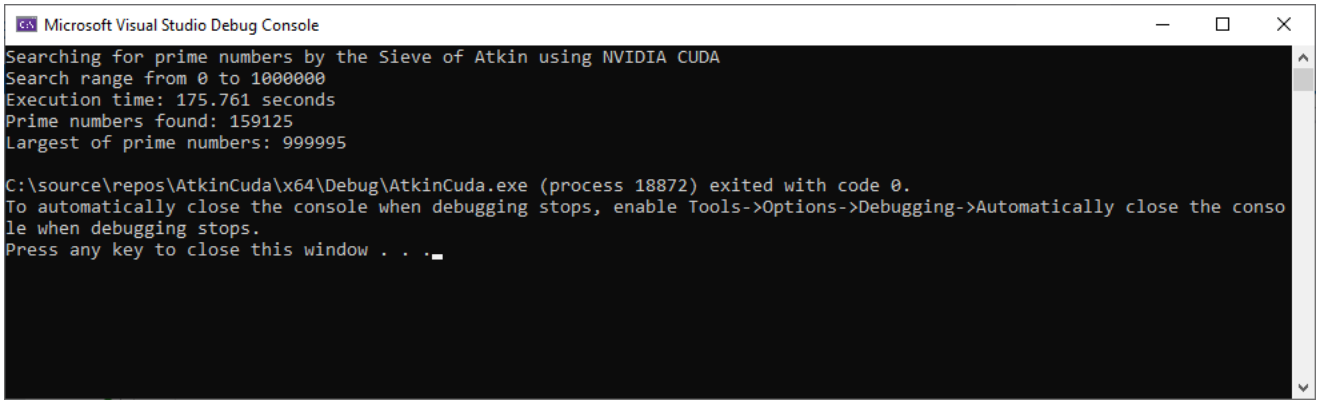


Рисунок 3.7 – Результати пошуку простих чисел у діапазоні 0÷1 000 000 решетом Аткіна з CUDA-реалізацією (лістинг у дод. Б)

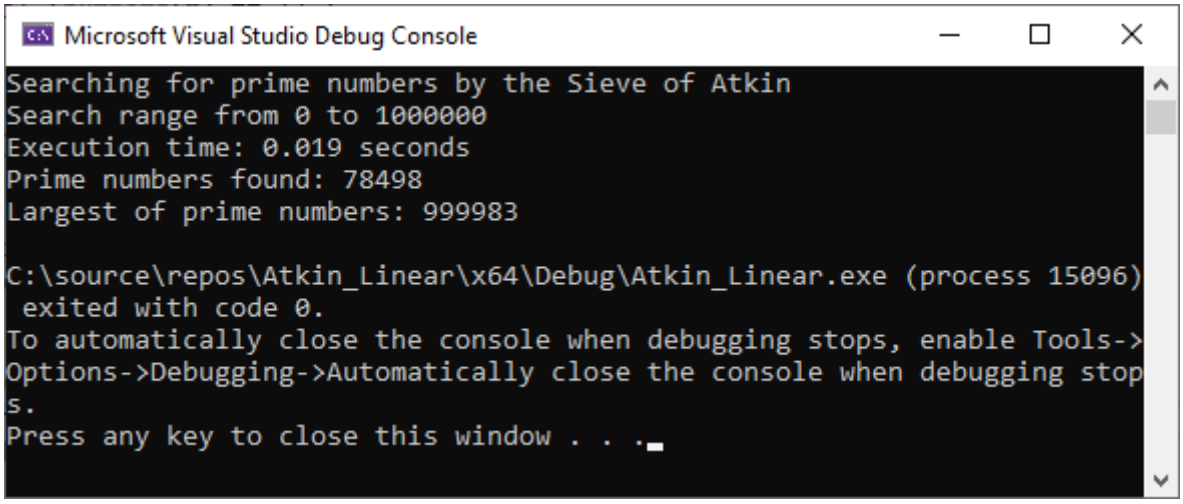


Рисунок 3.8 – Результати пошуку простих чисел у діапазоні 0÷1 000 000 решетом Аткіна з CPU-реалізацією

Також виконана спроба пошуку простих чисел у діапазоні до 10 000 000 чисел з використанням NVIDIA CUDA, але час очікування перебільшив 50 хвилин (рис. 3.9). Виконана спроба тестування CPU-реалізації решета Аткіна у діапазоні 0÷1000000000. Результати показали (рис. 3.10), що за 48 секунд алгоритм знайшов 53326268 чисел, хоча відомо, що таких чисел повинно бути 50847534, тобто майже

2,5 млн. з них не прости, а складені. Можна зробити висновок, що алгоритм містить помилки, але знайти їх не вдалося із-за неможливості трасування програми на таких великих діапазонах значень.

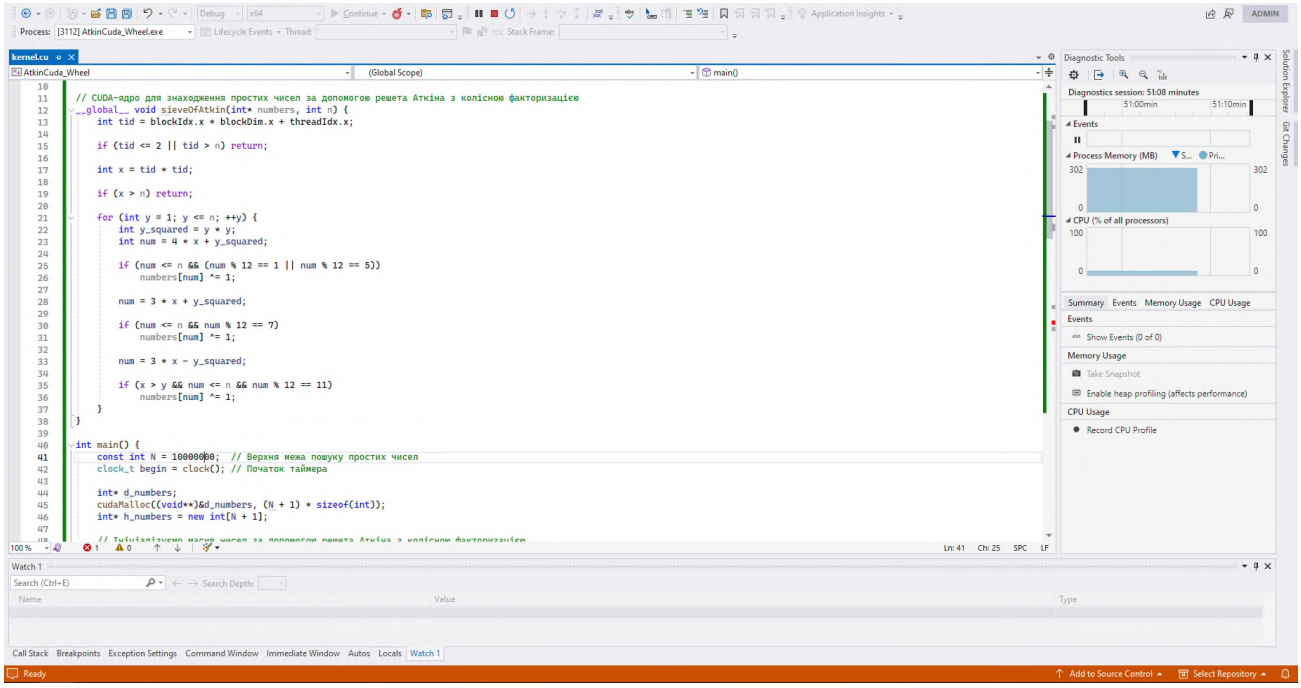


Рисунок 3.9 – Невдала спроба пошуку простих чисел у діапазоні 0÷10 000 000 решетом Аткіна з CUDA-реалізацією

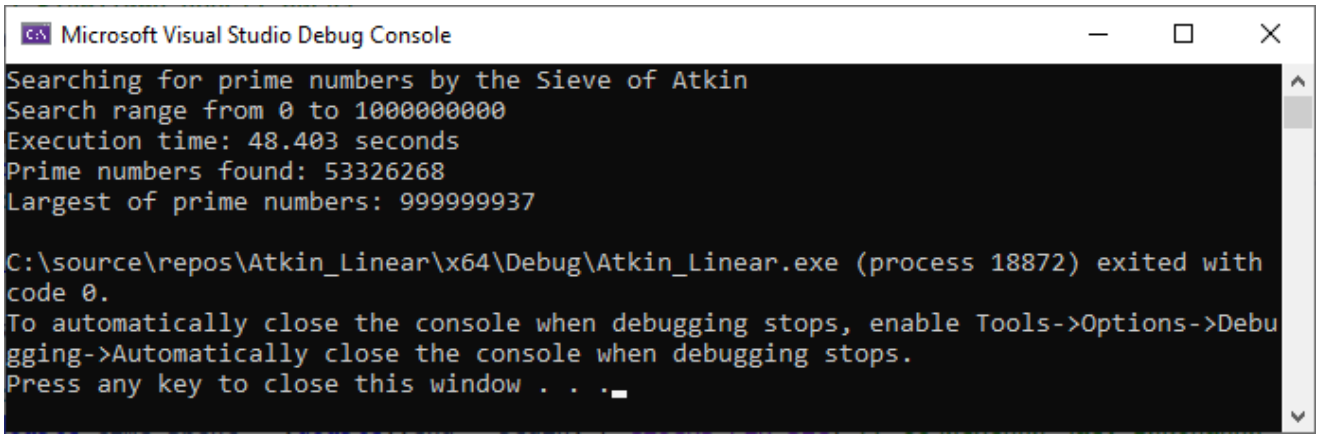


Рисунок 3.10 – Результати пошуку простих чисел у діапазоні 0÷1 000 000 000 решетом Аткіна з CPU-реалізацією з хибними значеннями

Виконано тестування програмної CPU-реалізації та CUDA-реалізації тесту Мілера-Рабіна для пошуку простих чисел. Основна ідея полягає в тому, щоб кожен CUDA-потік перевіряв власне число на простоту у заданому діапазоні. Всі

результати зберігаються в масиві булевих значень. Після закінчення обчислень цей масив повертається назад у host-частину програми для подальшого аналізу. Результати роботи на діапазоні $0 \div 1\,000\,000\,000$ показано на рис. 3.11-3.12.

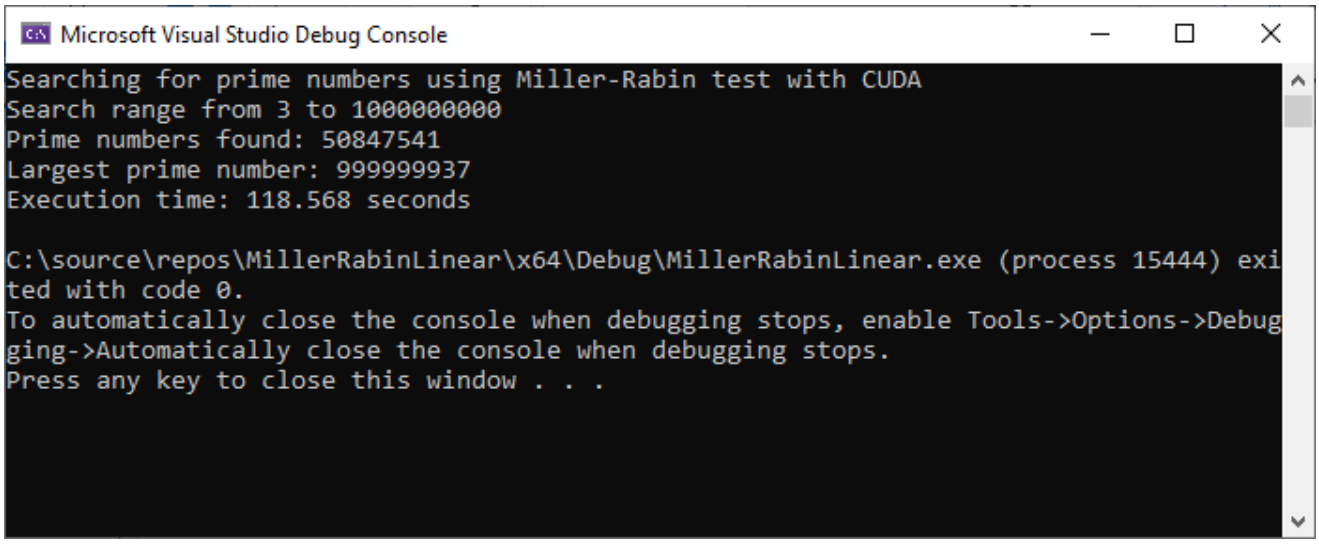


Рисунок 3.11 – Результати пошуку простих чисел у діапазоні 0÷1 000 000 000 тестом Мілера-Рабіна з CUDA-реалізацією (лістинг у дод. Б)

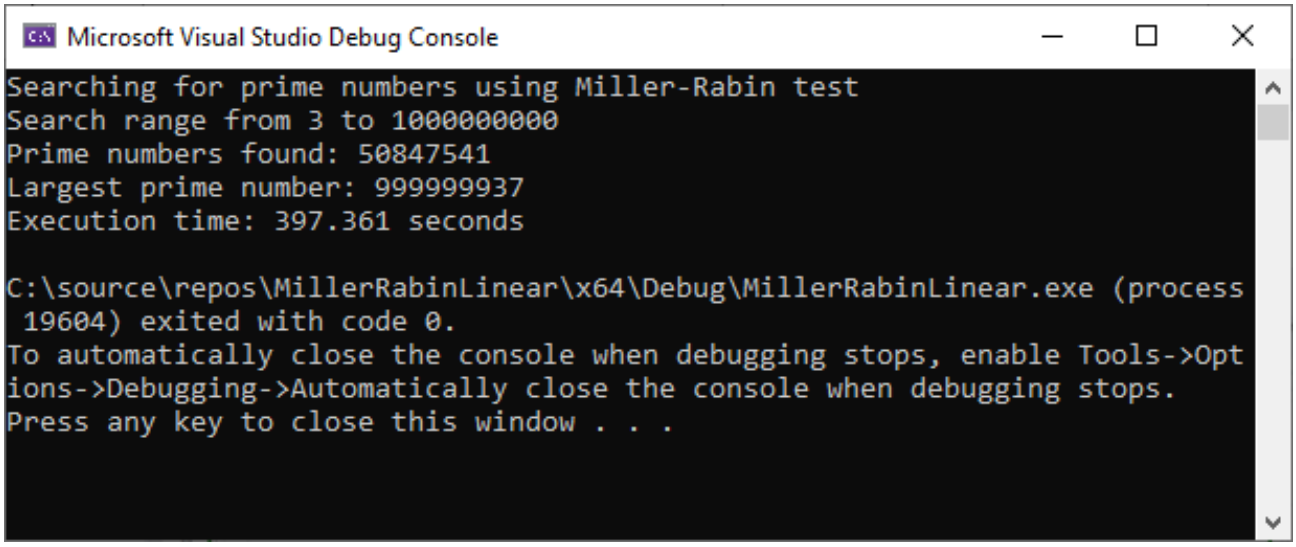


Рисунок 3.12 – Результати пошуку простих чисел у діапазоні 0÷1 000 000 000 тестом Мілера-Рабіна з CPU-реалізацією (лістинг у дод. Б)

Виконана тестування програмної CPU-реалізації та CUDA-реалізації алгоритму перевірки простоти чисел Мерсена, тобто простих чисел особливого виду $2^p - 1$, де p – натуральне число, за допомогою тесту Люка-Лемера. Результати роботи на діапазоні $0 \div 200\,000$ показано на рис. 3.13-3.14.

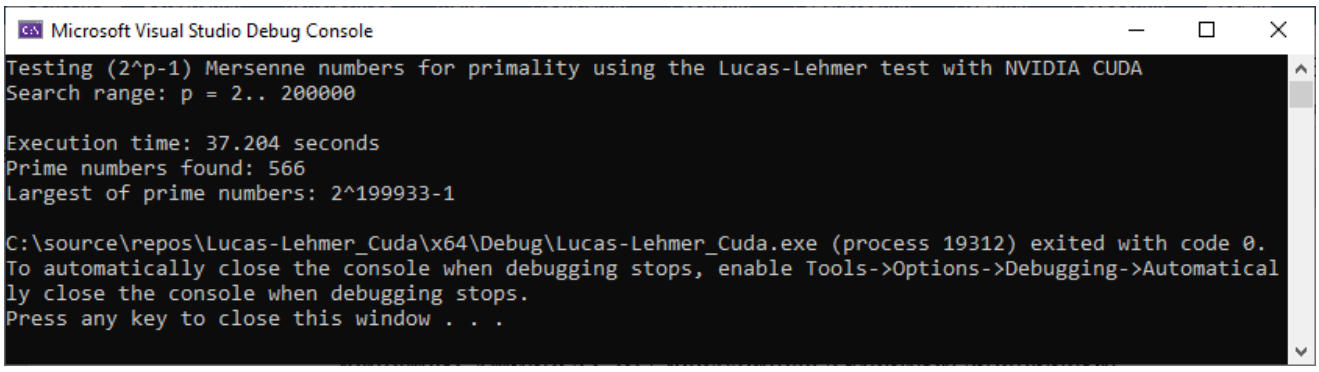


Рисунок 3.13 – Результати перевірки простоти чисел Мерсена тестом Люка-Лемера з CUDA-реалізацією (лістинг у дод. Б)

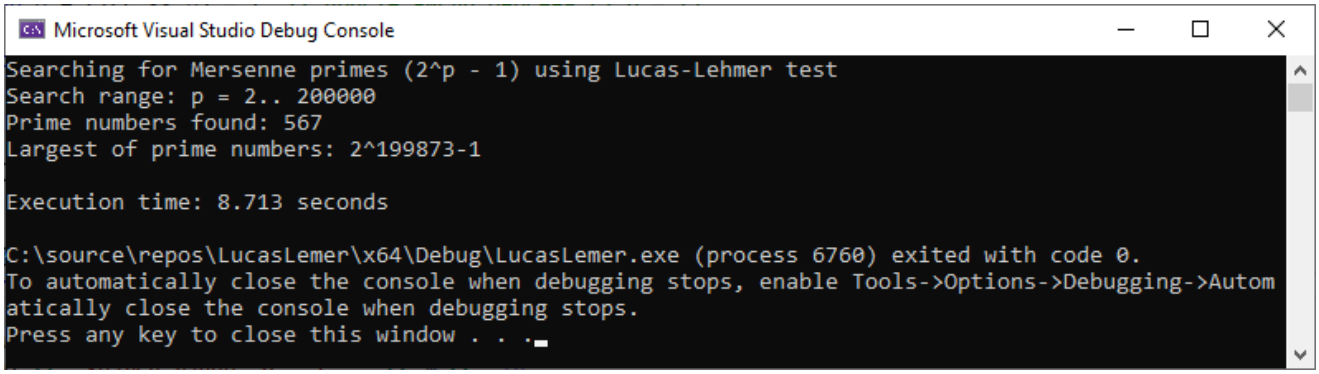


Рисунок 3.14 – Результати перевірки простоти чисел Мерсена тестом Люка-Лемера з CPU-реалізацією (лістинг у дод. Б)

3.2 Профілювання CUDA- та CPU-реалізацій алгоритмів пошуку простих чисел

Виконано профілювання CUDA- та CPU-реалізацій алгоритмів пошуку простих чисел у візуалізаторі MS Visualizer Concurrency, результати показано на рис. 3.15-3.19. У блоці «Visible Timeline Profile» наведена статистична інформація про навантаження на CPU.

Аналіз метрик продуктивності показав, що якщо порівняти навантаження на центральний процесор під час виконання CUDA-реалізації з виконанням CPU-реалізації, однозначно можна зробити висновок про зменшення навантаження в середньому на 46% (рис. 3.20). Найбільш вдалими CUDA-реалізаціями є решето Аткіна та тест Люка-Лемера, навантаження на CPU зменшено більш на 60%.

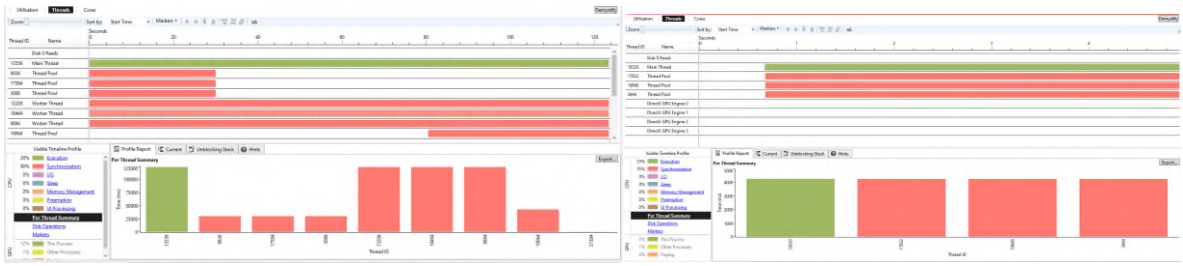


Рисунок 3.15 – Профілювання CUDA- та CPU-реалізації решета Ератосфена

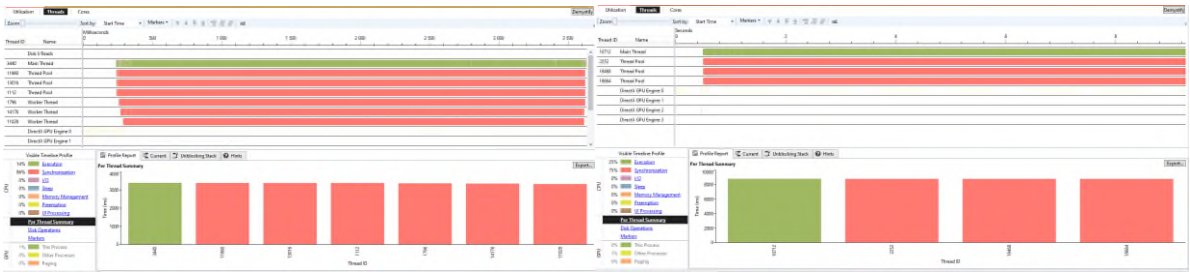


Рисунок 3.16 – Профілювання CUDA- та CPU-реалізації колісної факторизації

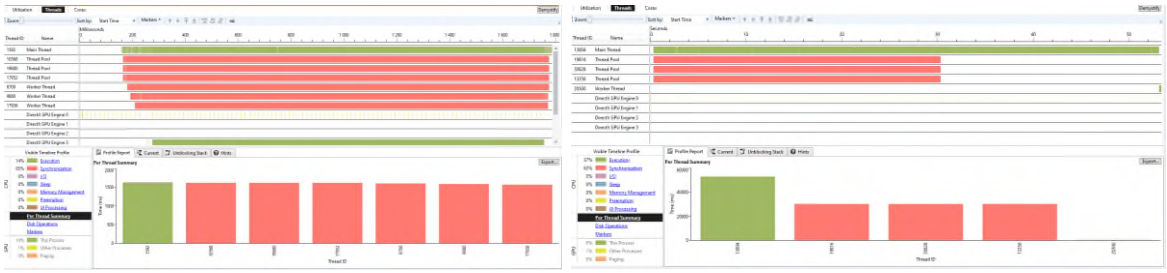


Рисунок 3.17 – Профілювання CUDA- та CPU-реалізації решета Аткіна

Рисунок 3.18 – Профілювання CUDA- та CPU-реалізації тесту Мілера-Рабіна

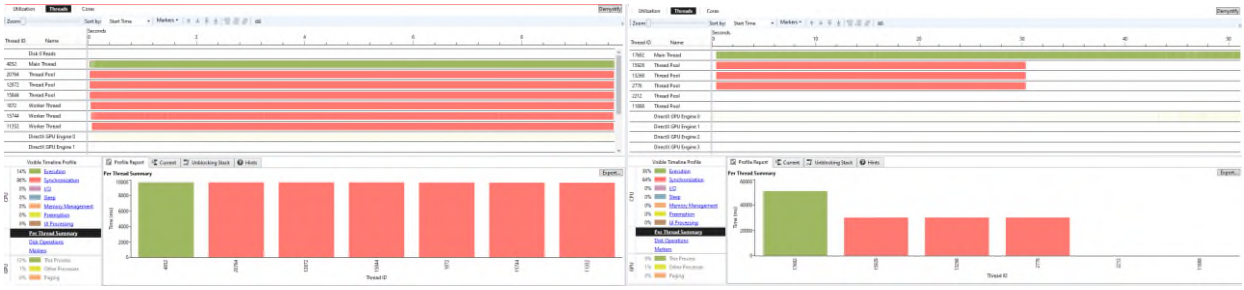


Рисунок 3.19 – Профілювання CUDA- та CPU-реалізації тесту Люка-Лемера

Рівень міжпотокової синхронізації CUDA-реалізацій алгоритмів достатньо високий (вище 80%), тому треба продовжувати вдосконалення алгоритмів з використанням прийомів мінімізації міжпроцесорних обмінів.

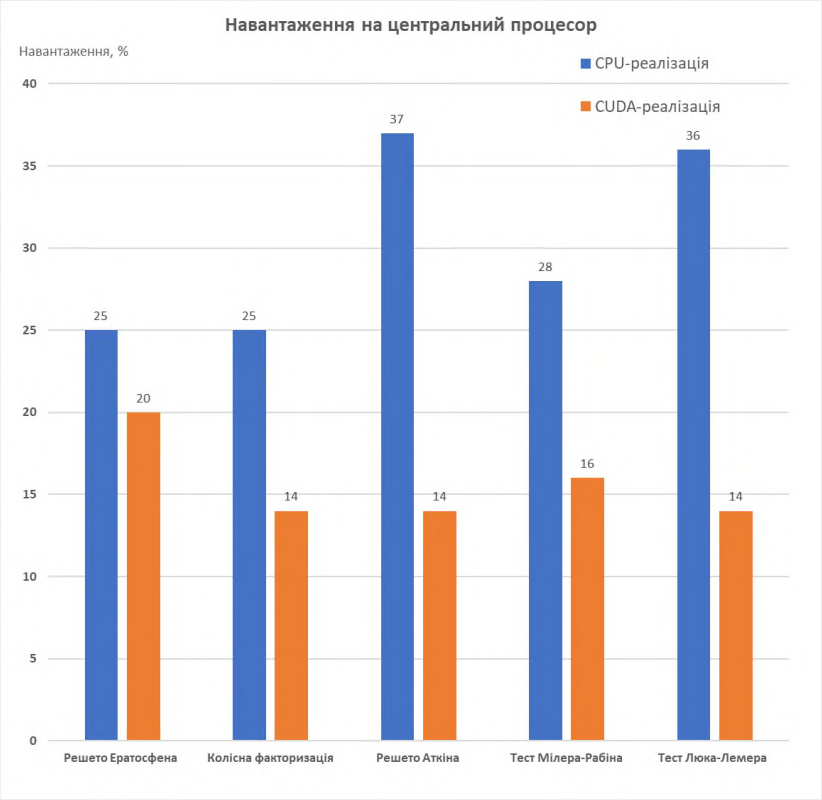


Рисунок 3.20 – Порівняння навантаження на центральний процесор

3.3 Оцінка ефективності CUDA-реалізацій алгоритмів пошуку простих чисел

Для оцінки ефективності CUDA-реалізацій проведено порівняння часу виконання та результатів пошуку простих чисел, які показано зокрема на рис. 3.2-3.14. Для більш зручного порівняння всі показники зібрано у табл. 3.1 та 3.2. Побудовано саме дві окремі таблиці, тому що процес пошуку чисел Мерсена за допомогою тесту Люка-Лемера суттєво відрізняється від інших алгоритмів, зокрема діапазонами пошуку. Порівняння часу виконання алгоритмів пошуку простих чисел для CUDA- та CPU-реалізацій показано на рис. 3.21-3.25.

Кількість знайдених простих чисел може відрізнятися, тому що в деяких алгоритмах пошук починався з числа 3, а в інших – з числа 1.

Таблиця 3.1 – Показники ефективності CPU- та CUDA-реалізацій алгоритмів пошуку простих чисел

Назва алгоритму	Межа пошуку	Час виконання	Знайдених простих чисел	Максимальне просте число
Решето Ератосфена, CPU-реалізація	1 000 000	0,91	78498	999983
	10 000 000	8,649	664579	9999991
	100 000 000	96,126	5761455	99999989
	1 000 000 000	946,543	50847534	999999937
Решето Ератосфена, CPU-реалізація паралельному режимі	1 000 000	0,12	78498	999983
	10 000 000	2,08	664579	9999991
	100 000 000	51,6	5761455	99999989
	1 000 000 000	1465,62	50847534	999999937
Решето Ератосфена, CUDA-реалізація	1 000 000	0,389	78500	999983
	10 000 000	1,457	664581	9999991
	100 000 000	12,004	5761457	99999989
	1 000 000 000	123,263	50847536	999999937
Колісна факторизація, CPU-реалізація	1 000 000	1,01	78498	999983
	10 000 000	8,835	664579	9999991
	100 000 000	91,634	5761455	99999989
	1 000 000 000	955,188	50847534	999999937
Колісна факторизація, CUDA-реалізація	1 000 000	0,456	78500	999983
	10 000 000	0,7	664581	9999991
	100 000 000	3,182	5761457	99999989
	1 000 000 000	29,437	50847536	999999937
Решето Аткіна, CPU-реалізація	1 000 000	0,02	78498	999983
	10 000 000	0,356	664579	9999991
	100 000 000	3,774	5761455	99999989
	1 000 000 000	48,403	53326268	999999937
Решето Аткіна, CUDA-реалізація	1 000 000	175,161	159191	999995
	10 000 000	null	null	null
	100 000 000	null	null	null
	1 000 000 000	null	null	null
Тест Мілера-Рабіна, CPU-реалізація	1 000 000	0,269	78498	999983
	10 000 000	3,048	664580	9999991
	100 000 000	34,405	5761458	99999989
	1 000 000 000	397,361	50847541	999999937
Тест Мілера-Рабіна, CUDA-реалізація	1 000 000	0,385	78498	999983
	10 000 000	1,429	664580	9999991
	100 000 000	12,057	5761458	99999989
	1 000 000 000	118,568	50847541	999999937

Таблиця 3.2 – Показники ефективності CPU- та CUDA-реалізацій алгоритмів пошуку чисел Мерсена тестом Люка-Лемера

Назва алгоритму	Межа пошуку	Час виконання	Знайдених простих чисел	Максимальне просте число
Тест Люка-Лемера, CUDA-реалізація	100 000	9,631	307	$2^{99839}-1$
	200 000	37,204	566	$2^{199373}-1$
	300 000	70,717	818	$2^{299969}-1$
	400 000	123,12	1055	$2^{399937}-1$
	500 000	219,92	1294	$2^{499969}-1$
Тест Люка-Лемера, CPU-реалізація	100 000	2,42	308	$2^{99713}-1$
	200 000	8,713	567	$2^{199873}-1$
	300 000	18,862	819	$2^{299969}-1$
	400 000	32,861	1057	$2^{399937}-1$
	500 000	50,47	1293	$2^{499969}-1$

Порівняння часу виконання реалізації решета Ератосфена у послідовному варіанті мовою C++ та з використанням бібліотеки паралельного програмування TPL мовою C# показало, що паралелізм дозволяє досягнути кращих результатів, але при зростанні діапазону пошуку ситуацію змінюється на протилежну (рис. 3.21). Результати CUDA-реалізації демонструють стабільне зменшення часу виконання при збільшені діапазону пошуку у середньому на 78%. Можна обґрунтовано стверджувати, що CUDA-реалізація решета Ератосфена повністю себе виправдовує.

Результати CUDA-реалізації колісної факторизації демонструють значне зменшення часу виконання при збільшені діапазону пошуку у середньому на 85%, при чому зафіксовано зменшення часу виконання до 97% (рис. 3.22). Можна обґрунтовано стверджувати, що CUDA-реалізація колісної факторизації показала найкращі результати.

Результати CUDA-реалізації решета Аткіна на відміну від попередніх алгоритмів показали погані результати. Результати вдалося отримати лише для діапазону не більш 1000000 чисел, при збільшенні діапазону час очікування встановити не вдалося (рис. 3.23). Крім того час виконання значно поступається всім іншим реалізаціям, а кількість хибних спрацювань перебільшує 50%.

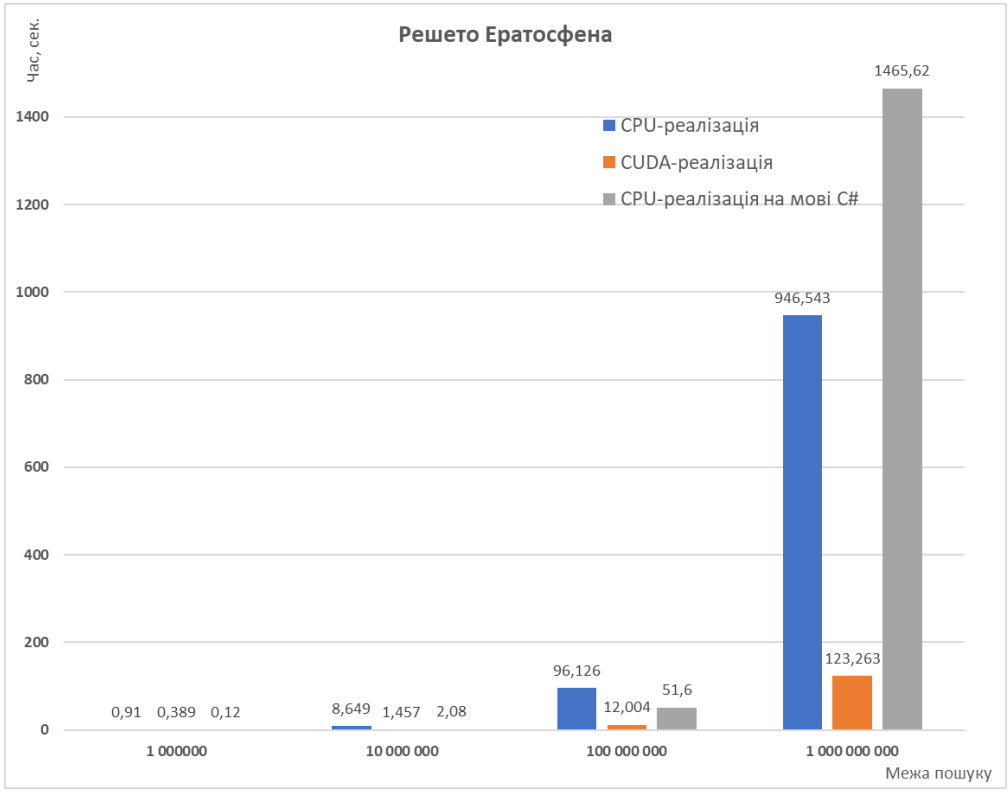


Рисунок 3.21 – Порівняння часу виконання алгоритмів пошуку простих чисел решетом Ератосфена

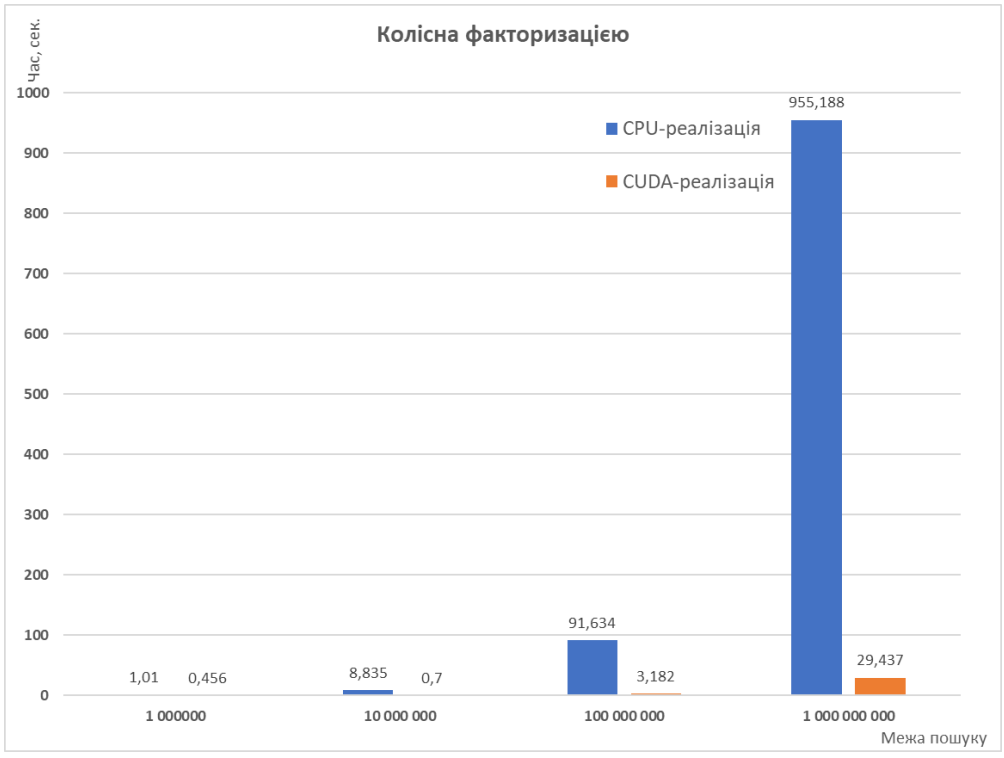


Рисунок 3.22 – Порівняння часу виконання алгоритмів пошуку простих чисел з колісною факторизацією

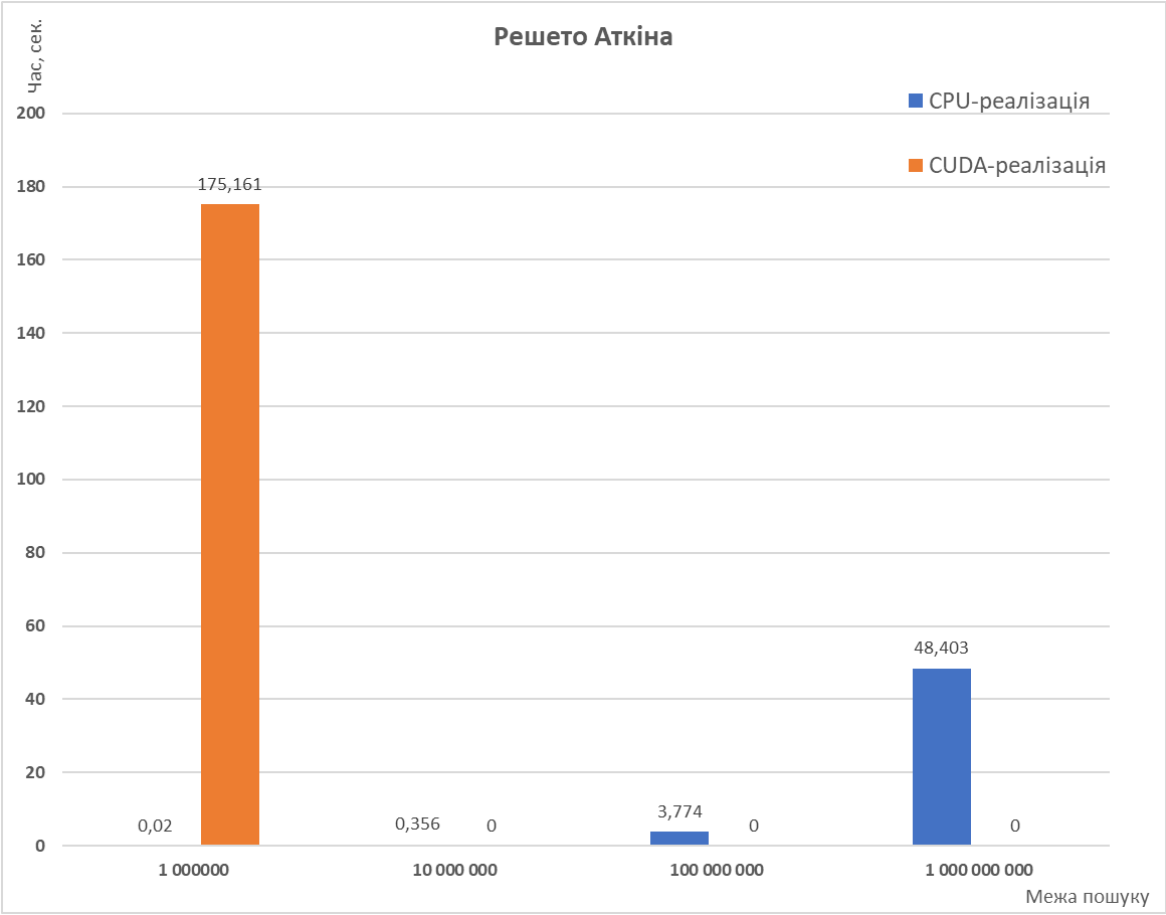


Рисунок 3.23 – Порівняння часу виконання алгоритмів пошуку простих чисел решетом Аткіна

Знайти помилку у CUDA-програмі не вдалося у зв'язку з відсутність доступу до середовища виконання CUDA та неможливістю її трасування. Можна зробити висновок, що CUDA-реалізація решета Аткіна повністю провалена.

Результати CUDA-реалізації пошуку простих чисел тестом Мілера-Рабіна демонструють зменшення часу виконання, починаючі з діапазону пошуку у 10000000 чисел, в середньому на 62% (рис. 3.24). Можна обґрунтовано стверджувати, що CUDA-реалізація тесту Мілера-Рабіна теж себе виправдовує.

Порівняння результатів виконання CPU- та CUDA-реалізацій пошуку чисел Мерсена тестом Люка-Лемера демонструють не зменшення, а збільшення зменшення часу виконання CUDA-реалізацій в середньому на 63% (рис. 3.25). Можна обґрунтовано стверджувати, що CUDA-реалізація тесту Люка-Лемера провалена. Погані результати можна пояснити неоптимальною CUDA-реалізацією тесту Люка-Лемера.

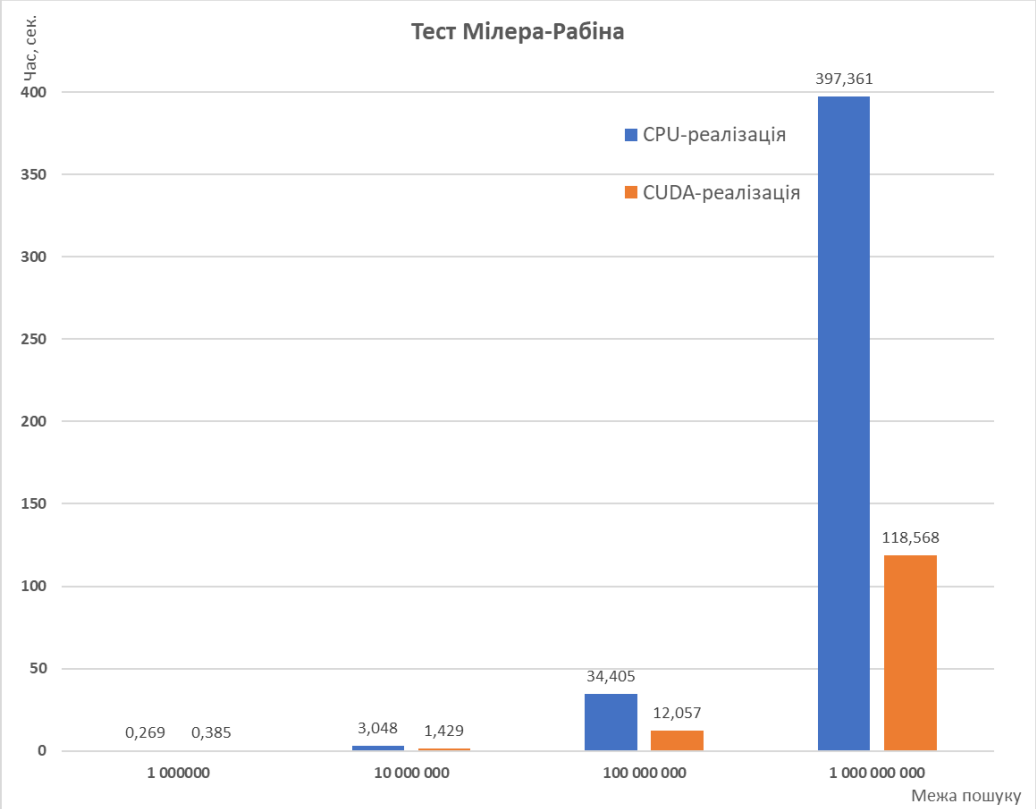


Рисунок 3.24 – Порівняння часу виконання алгоритмів пошуку простих чисел тестом Мілера-Рабіна

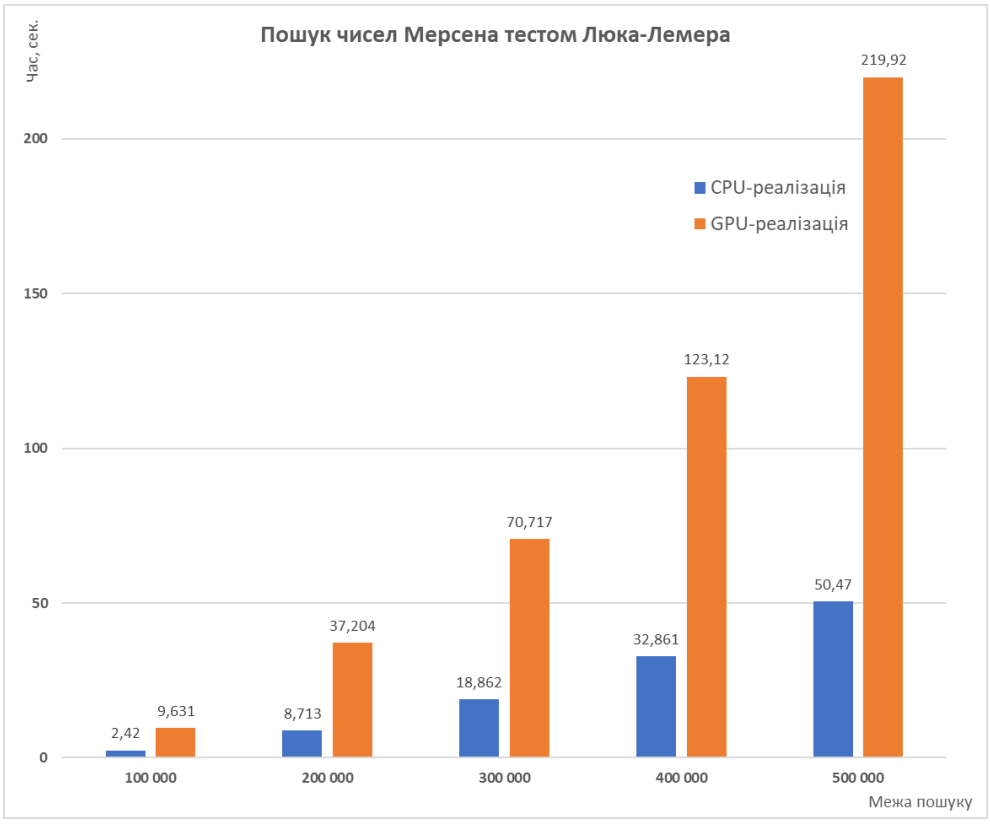


Рисунок 3.25 – Порівняння часу виконання алгоритмів пошуку чисел Мерсена тестом Люка-Лемера

3.4 Висновки за розділом

В результаті аналізів результатів тестування та профілювання програмних CPU- та CUDA-реалізацій різних алгоритмів пошуку простих чисел встановлено, що використання CUDA-технології для рішення складних обчислювальних задач, зокрема для пошуку простих чисел за допомогою решета Ератосфена, колісної факторизації та тесту Мілера-Рабіна, значно зменшує час виконання та навантаження на центральний процесор, тому заслуговує подальших досліджень.

Зафіксовано стабільне зменшення часу виконання пошуку у середньому на 78% для решета Ератосфена, на 85% для колісної факторизації, на 62% для тесту Мілера-Рабіна.

Аналіз метрик продуктивності дозволив зробити висновок про зменшення навантаження в середньому на 46%. Найбільш вдалими в даному випадку CUDA-реалізаціями є решето Аткіна та тест Люка-Лемера, при чому навантаження на CPU зменшено більш на 60%.

Рівень міжпотокової синхронізації CUDA-реалізацій алгоритмів становить вище 80%, тому треба продовжувати вдосконалення алгоритмів з використанням прийомів мінімізації міжпроцесорних обмінів.

До невдалих результатів можна віднести факт, що CUDA-реалізація решета Аткіна та тесту Люка-Лемера показала значне зростання часу виконання у порівнянні зі звичайною CPU-реалізацією.

ВИСНОВКИ

В роботі виконано завдання програмної реалізації та оцінки ефективності процесу пошуку простих чисел засобами NVIDIA CUDA.

В результаті аналізу предметної галузі встановлено, що:

– сучасні програмно-апаратні платформи паралельної обробки даних широко використовують графічні процесори для прискорення складних обчислень;

– популярні алгоритми пошуку простих чисел, зокрема решето Ератосфена, колісна факторізація, решето Аткіна, тести Люка-Лемера та Мілера-Рабіна, можуть бути реалізовані за допомогою доступних програмно-апаратних технологій;

– для оцінки продуктивності застосунків можна використати візуалізатор паралелізму MS Visualizer Concurrency.

Під час проектування отримано наступні результати:

– розроблено блок-схеми алгоритмів знаходження простих чисел в заданому діапазоні засобами CUDA для паралельного обчислення на графічних пристроях, а саме: решето Ератосфена, колісна факторизація, решето Аткіна, тест Міллера-Рабіна, а також алгоритм знаходження простих чисел Мерсена за допомогою тесту Люка-Лемера;

– для програмної реалізації алгоритмів використано програмно-апаратну платформу NVIDIA CUDA, IDE Visual Studio та мову програмування C++.

В результаті аналізів результатів тестування та профілювання програмних CPU- та CUDA-реалізацій різних алгоритмів пошуку зафіксовано стабільне зменшення часу виконання пошуку у середньому на 78% для решета Ератосфена, на 85% для колісної факторизації, на 62% для тесту Мілера-Рабіна. Аналіз метрик продуктивності дозволив зробити висновок про зменшення навантаження в середньому на 46%. Найбільш вдалими CUDA-реалізаціями є решето Аткіна та тест Люка-Лемера, навантаження на CPU зменшено більш на 60%.

Рівень міжпотокової синхронізації CUDA-реалізацій алгоритмів досягає

80%, що свідчить про погану реалізацію міжпроцесорних та міжпотокових обмінів даними у середовищі CUDA.

До невдалих результатів можна віднести факт, що CUDA-реалізація решета Аткіна та тесту Люка-Лемера показала значне зростання часу виконання у порівнянні зі звичайною CPU-реалізацією.

У напрямку подальших досліджень необхідно розглянути питання мінімізації міжпроцесорних та міжпотокових обмінів у CUDA-пристрої, збільшення ступеню паралелізму алгоритмів пошуку та оптимізації організації даних для запобігання процесам переповнення буферу, зокрема використання типу даних «biginteger».

Апробація кваліфікаційної роботи виконана на VI Всеукраїнській науково-практичній інтернет-конференції «Сучасні технології в енергетиці, електромеханіці, системах управління та машинобудуванні» (м. Харків, 06-07 грудня 2023 р.) з публікацією тези доповіді «Organization of prime number search process using NVIDIA CUDA».

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Трофименко О. Г., Логінова Н. І., Манаков С. Ю. Методичні вказівки до виконання кваліфікаційної роботи здобувачами першого (бакалаврського) рівня вищої освіти за спеціальностями 121 «Інженерія програмного забезпечення» та 122 «Комп'ютерні науки» [Електронне видання] / О. Г. Трофименко, Н. І. Логінова, С. Ю. Манаков. – Одеса, 2024. – 39 с.

2. Vandana B. Parallelization of corner sort with CUDA for many-objective optimization. *Proceedings of the Genetic and Evolutionary Computation Conference*. 2022.

3. Qiao S. Evolving the HPL benchmark towards multi-GPGPU clusters. *CCF Transactions on High Performance Computing 5.1*. 2023. P. 84-96.

4. Сорокін М. Розпаралелювання чисельних розв'язків рівнянь мілкої води методом скінченних об'ємів для реалізації на багатопроцесорних системах і графічних процесорах. *Екологічна безпека та природокористування*. 46(2), 2023. С. 163-193.

5. Chykunov P. Organization of prime number search process using NVIDIA CUDA / P. Chykunov, V. Chernetskyi // Матеріали VI Всеукраїнської науково-практичної інтернет-конференції (м. Харків, 06-07 грудня 2023 р.): *Сучасні технології в енергетиці, електромеханіці, системах управління та машинобудуванні.* – Харків: ННППІ УІПА, 2023. С. 40-41.

6. Гришанович Т. О. Лабораторний практикум із дисципліни "Паралельні та розподілені обчислення" [Електронний ресурс]; ВНУ імені Лесі Українки. Луцьк : ВНУ імені Лесі Українки, 2022. 50 с.

7. Минайленко Р. М. Паралельні та розподілені обчислення : навч. посіб. М-во освіти і науки України, Центральноукраїн. нац. техн. ун-т. – Кропивницький: ЦНТУ, 2021. – 153 с.

8. CUDA Programming and Performance. URL: https://forums.developer.nvidia.com/c/accelerated-computing/cuda/cuda-programming-and-performance/7.

9. AMD Accelerated Parallel Processing. URL: https://community.amd.com/t5/archives-discussions/ati-stream-sdk-is-now-amd-accelerated-parallel-processing-app/m-p/93462.

10. Онлайн-курс Udemy «Multithreading and Parallel Programming in C#». URL: https:// www.udemy.com/course/parallel-csharp.

11. Wiki: Решето Ератосфена. URL: https://uk.wikipedia.org/wiki/ Решето_Ератосфена.

12. Wiki: Wheel factorization. URL: https://en.wikipedia.org/wiki/Wheel_factorization.

13. Wiki: Решето Аткіна. URL: https://uk.wikipedia.org/wiki/ Решето Аткіна.

14. Wiki: Тест Люка-Лемера. URL: https://wikipedia.org/wiki/Тест_Люка_—_Лемера.

15. Wiki: Тест простоти Ферма. URL: https://uk.wikipedia.org/wiki/Тест_простоти_Ферма.

16. Wiki: Тест простоти Міллера-Рабіна. URL: https://uk.wikipedia.org/wiki/ Тест_простоти_Міллера_—_Рабіна.

17. Top500, November 2023. URL: https://www.top500.org/lists/top500/2023/11.

18. Indian government launches $1.2bn IndiaAI mission. URL: https://www.datacenterdynamics.com/en/news/indian-government-launches-12bn-indiaai-mission-plans-10000-gpu-supercomputer.

19. NVIDIA HPC Application Performance. URL: https://developer.nvidia.com/hpc-application-performance.

20. Great Internet Mersenne Prime Search. URL: https://www.mersenne.org.

21. Parallel programming in .NET: A guide to the documentation – Microsoft Learn. URL: https://learn.microsoft.com/en-us/dotnet/standard/parallel-programming.

22. George B., Nagpal P. Optimizing parallel applications using concurrency visualizer: A case study. Parallel Computing Platform Group, Microsoft Corporation, 2010.

23. Peternier A. et al. Parallelism profiling and wall-time prediction for multi-

threaded applications. *In: Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering*. 2013. p. 211-216.

24. Чикунов П. О. Можливості мови C# для організації паралельних обчислень / Матеріали міжнар. наук.-практ. конф: *Кіберпростір в умовах війни та глобальних викликів XXI століття: теорія та практика*. Одеса, 2023. – С. 140-143.

25. Chykunov P. Profiling multithreaded programs in the Visualizer Concurrency / P. Chykunov, A. Kotov // Матеріали VI Всеукр. наук.-практ. інтернет-конф.: *Сучасні технології в енергетиці, електромеханіці, системах управління та машинобудуванні*: Харків: ННППІ УІПА, 2023. С. 36-38.

26. Чикунов П.О. Програмна реалізація основних операторів генетичних алгоритмів / П.О. Чикунов, П.О. Дмітрієв, А.В. Мурашко // Матеріали II Всеукр. наук.-практ. інтернет-конф.: *Сучасні технології в енергетиці, електромеханіці, системах управління та машинобудуванні*. Бахмут: ННППІ УІПА, 2019. – С. 32.

27. Яровий А. А., Кулик О. О., Арсенюк І. Р. Комп'ютерне моделювання процесу паралельного оброблення зображень на основі технологій OpenMP та NVIDIA CUDA. *Методи та системи оптико-електронної і цифрової обробки зображень та сигналів,* 2015.

ДОДАТКИ

Додаток А. Апробація результатів роботи

ORGANIZATION OF PRIME NUMBER SEARCH PROCESS USING NVIDIA CUDA

Chykunov P., Candidate of Technical Sciences, Associate Professor,
Chernetskyi V., bachelor level,
National University "Odesa Law Academy"

Recently, several technologies have become available to developers for organizing and supporting parallel programming. One such technology is GPGPU, which enables general-purpose computing on the graphics processing units (GPUs) of regular discrete video cards.

For developers using the Visual Studio IDE and programming in C and C++, NVIDIA, the video adapter manufacturer, offers the proprietary CUDA platform (Compute Unified Device Architecture), which works with the GeForce series of GPUs. CUDA supports multiple programming languages, including C, C++, Fortran, and Python, providing programmers with a convenient toolkit. NVIDIA also provides a range of libraries for CUDA, such as cuBLAS (linear algebra library), cuFFT (fast Fourier transform library), cuDNN (deep learning library), and many others, simplifying the creation of task-specific programs.

CUDA supports various models and series of NVIDIA GPUs, allowing developers to create programs that run on a wide range of hardware. Organizing code for CUDA requires careful memory management and proper utilization of blocks and threads. Additionally, optimizations can be applied to enhance performance, such as using shared memory and optimizing global memory access.

The authors set out to utilize available hardware and software resources to support the process of organizing parallel computations for well-known complex algorithms, such as the Sieve of Eratosthenes for prime number searching.

Typically, CUDA programs consist of main code executed on the central processing unit (CPU) and CUDA cores executed on the GPU. CUDA cores can be optimized for parallel execution of specific tasks. CUDA provides access to various types of GPU memory, including global memory, constant memory, texture memory, and shared memory.

To implement the Sieve of Eratosthenes using NVIDIA CUDA, the following steps are necessary:

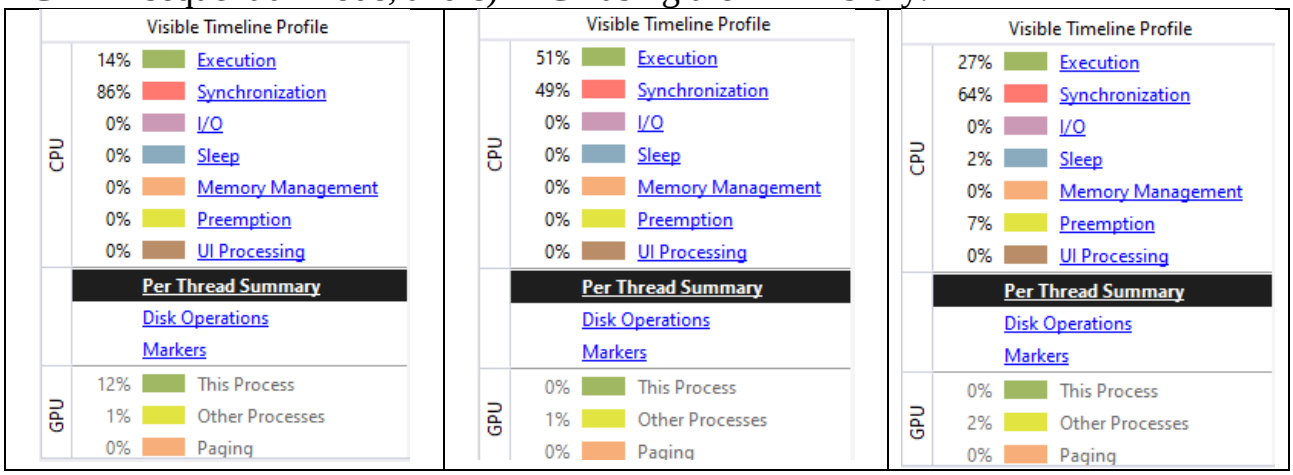
1. Divide the range of numbers to find primes into smaller intervals. Each interval will be processed by a separate CUDA block.
2. Choose an algorithm for prime numbers, such as the Sieve of Eratosthenes.
3. Distribute subintervals of numbers among CUDA threads.
4. Use CUDA synchronization mechanisms like "syncthreads" to avoid collisions and correctly combine results.

5. Organize the process of handling computation results on the CPU.

6. Utilize CUDA optimization techniques, such as shared memory for data exchange within a block, to improve computation speed.

7. After obtaining results from different number ranges and verifying them using guides, analyze computation performance using the Concurrency Visualizer parallelism visualizer.

The authors developed three versions of the application for prime number searching up to one hundred million: a) in C++ using the NVIDIA CUDA platform, b) in C++ in sequential mode, and c) in C# using the TPL library.



a) NVIDIA CUDA b) serial mode c) multithreading TPL

Figure 1 – Comparison of performance metrics of parallel and sequential algorithms for searching prime numbers

The analysis of performance metrics obtained through the Concurrency Visualizer revealed that when using NVIDIA CUDA, the process took 12 seconds with the following load: CPU 14%, GPU 12%. In sequential mode, the process took 96 seconds with the following load: CPU 51%, GPU 0%. We achieved a significant acceleration (8 times) in the prime number search process using NVIDIA CUDA. The multithreaded C# application version executed in only 2 seconds with a CPU load of 27%, GPU 0%. Thus, the application of multithreaded programming significantly improved computational performance, even though there is no TPL library version for C++. This can be explained by the fact that the CPU power on the research computer significantly exceeds the GPU power and the absence of CUDA optimization mechanisms.

It can be reasonably argued that the use of GPGPU technology for solving complex computational tasks greatly accelerates computational processes and deserves further research.

List of References

1. Vandana B. Parallelization of corner sort with CUDA for many-objective optimization. Proceedings of the Genetic and Evolutionary Computation Conference. 2022.

2. Qiao S. Evolving the HPL benchmark towards multi-GPGPU clusters. CCF Transactions on High Performance Computing 5.1, 2023. P. 84-96.

Додаток Б. Програмний код

Лістинг файлу Eratosfen_Linear.cpp пошуку простих чисел решетом Ератосфена з CPU-реалізацією

```
#include <iostream>
#include <vector>
#include <cmath>
#include <ctime>

// Функція для пошуку простих чисел за допомогою решета Ератосфена
std::vector<int> sieveOfEratosthenes(int n) {
    std::vector<bool> prime(n + 1, true); // Ініціалізуємо вектор для запам'ятовування
простих чисел
    std::vector<int> primes; // Вектор для зберігання знайдених простих чисел

    // Застосовуємо решето Ератосфена
    for (int p = 2; p * p <= n; ++p) {
        if (prime[p]) {
            for (int i = p * p; i <= n; i += p) {
                prime[i] = false; // Викреслюємо всі кратні числа
            }
        }
    }
    // Заповнюємо вектор простими числами
    for (int p = 2; p <= n; ++p) {
        if (prime[p]) {
            primes.push_back(p);
        }
    }
    return primes;
}

int main() {
    const int N = 1000000; // Верхня межа пошуку простих чисел

    std::cout << "Searching for prime numbers using the Sieve of Eratosthenes\n";
    std::cout << "Search range from 2 to " << N << "\n";

    clock_t begin = clock(); // Початок вимірювання часу

    // Знаходимо прості числа за допомогою решета Ератосфена
    std::vector<int> primes = sieveOfEratosthenes(N);

    clock_t end = clock(); // Кінець вимірювання часу
    // Обчислення часу виконання
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    std::cout << "Execution time: " << time_spent << " seconds\n";

    // Виведення кількості знайдених простих чисел
    std::cout << "Prime numbers found: " << primes.size() << "\n";

    // Виведення найбільшого простого числа
    if (!primes.empty()) {
        std::cout << "Largest prime number: " << primes.back() << "\n";
    }
    else {
        std::cout << "No prime numbers found\n";
    }

    return 0;
}
```

Лістинг файлу EratosfenCuda.cu пошуку простих чисел решетом Ератосфена з CUDA-реалізацією

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <nlohmann/json.hpp>
// CUDA-ядро для знаходження простих чисел за допомогою решета Ератосфена
__global__ void sieveOfEratosthenes(int* numbers, int n) {
    // Отримуємо глобальний ідентифікатор потоку
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    // Виключаємо 0 і 1 з решета
    if (tid >= 2 && tid <= n) {
        // Якщо число не викреслене, то викреслюємо його кратні
        if (numbers[tid] == 1) {
            for (int i = 2 * tid; i <= n; i += tid) {
                numbers[i] = 0;
            }
        }
    }
}
// структура для збереження простих чисел
struct PrimeData
{
    int count;
    int max;
    std::vector<int> primes;
};

int main() {
    const int N = 1000000000;  // Задаємо верхню межу пошуку простих чисел
    // Запускаємо таймер
    double time_spent = 0.0;
    clock_t begin = clock();

    // Виділяємо пам'ять під масив чисел та копіюємо його на пристрій
    int* d_numbers;
    cudaMalloc((void**)&d_numbers, (N + 1) * sizeof(int));
    int* h_numbers = new int[N + 1];
    // Ініціалізуємо масив чисел
    for (int i = 0; i <= N; ++i) {
        h_numbers[i] = 1;  // Спочатку вважаємо всі числа простими
    }
    // Копіюємо масив чисел на пристрій
    cudaMemcpy(d_numbers, h_numbers, (N + 1) * sizeof(int), cudaMemcpyHostToDevice);
    // Задаємо параметри сітки та блоку потоків
    int blockSize = 256;
    int gridSize = (N + blockSize - 1) / blockSize;
    // Запускаємо CUDA-ядро для пошуку простих чисел
    sieveOfEratosthenes << <gridSize, blockSize >> > (d_numbers, N);

    // Копіюємо результат назад на хост
    cudaMemcpy(h_numbers, d_numbers, (N + 1) * sizeof(int), cudaMemcpyDeviceToHost);
    clock_t end = clock();
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;
    // Виводимо діапазон пошуку та час виконання
    std::cout << "Searching for prime numbers by the Sieve " +
                 "of Eratosthenes using NVIDIA CUDA" << std::endl;
    std::cout << "Search range from 0 to " << N << std::endl;
    std::cout << "Execution time: " << time_spent << " seconds" << std::endl;
    // Встановлюємо кількість знайдених простих чисел
    std::cout << "Prime numbers found: ";
```

```
    int Count_of_Primes = 0;
    int Max_of_Primes = 0;
    PrimeData primeData;

    for (int i = 0; i <= N; ++i) {
        if (h_numbers[i] == 1)
        {
            Count_of_Primes++;
            Max_of_Primes = i;
            primeData.primes.push_back(i);
        }
    }
    primeData.count = Count_of_Primes;
    primeData.max = Max_of_Primes;
    // створюємо структуру файлу у форматі Jason
    nlohmann::json jsonData;
    jsonData["count"] = primeData.count;
    jsonData["max"] = primeData.max;
    jsonData["primes"] = primeData.primes;
    // Відкриваємо файл для запису
    std::ofstream outFile("primes.json");
    // Записуємо дані у файл
    outFile << std::setw(4) << jsonData << std::endl;
    // Закриваємо файл
    outFile.close();
    // Вивід кількості простих чисел у консоль
    std::cout << Count_of_Primes << std::endl;
    // Вивід максимального простого числа у консоль
    std::cout << "Largest of prime numbers: " << Max_of_Primes << std::endl;
    // Звільняємо пам'ять
    delete[] h_numbers;
    cudaFree(d_numbers);

    return 0;
}
```

Лістинг файлу EratosfenTrheads.cs пошуку простих чисел решетом Ератосфена з CPU-реалізацією

```
using System.Threading.Tasks;
class Program
{
    static void Main(string[] args)
    {
        int start = 2;
        int end = 100000000;
        int range = (end - start + 1) / 12; // Розділяємо діапазон на 12 частин

        Console.WriteLine($"Searching for prime numbers using the Sieve of
Eratosthenes");
        Console.WriteLine($"Search range from 2 to {end}");
        Task[] tasks = new Task[12];
        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();
        int totalPrimes = 0;
        int lastPrime = 0;
        for (int i = 0; i < 12; i++)
        {
            int delegateStart = start + i * range;
            int delegateEnd = (i == 11) ? end : delegateStart + range - 1;
            tasks[i] = Task.Run(() =>
            {
                var (primes, last) = FindPrimes(delegateStart, delegateEnd);
```

```
                totalPrimes += primes;
                lastPrime = last;
            });
        }
        Task.WaitAll(tasks);
        stopwatch.Stop();
        double elapsedTimeInSeconds = stopwatch.Elapsed.TotalSeconds;
        Console.WriteLine($"Execution time: {elapsedTimeInSeconds} seconds.");
        Console.WriteLine($"Prime numbers found: {totalPrimes}");
        Console.WriteLine($"Largest prime number: {lastPrime}");
    }

    static (int, int) FindPrimes(int start, int end)
    {
        int count = 0;
        int lastPrime = 0;
        for (int i = start; i <= end; i++)
        {
            bool isPrime = true;
            for (int j = 2; j <= Math.Sqrt(i); j++)
            {
                if (i % j == 0)
                {
                    isPrime = false;
                    break;
                }
            }
            if (isPrime)
            {
                count++;
                lastPrime = i;
            }
        }
        return (count, lastPrime);
    }
}
```

Лістинг файлу Wheel_factorizationCuda.cu для пошуку простих чисел з колісною факторизацією та CUDA-реалізацією

```
#include <cuda_runtime.h>
#include "device_launch_parameters.h"
#define BLOCK_SIZE 256

// CUDA-ядро для знаходження простих чисел за допомогою решета Ератосфена з колісною
факторизацією
__global__ void sieveOfEratosthenes(int* numbers, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

    if (tid >= 2 && tid <= n) {
        if (numbers[tid] == 1) {
            for (int i = tid * tid; i <= n; i += tid) {
                numbers[i] = 0;
            }
        }
    }
}

int main() {
    const int N = 1000000000;  // Верхня межа пошуку простих чисел
    clock_t begin = clock(); // Початок таймера

    int* d_numbers;
```

```
    cudaMalloc((void**)&d_numbers, (N + 1) * sizeof(int));
    int* h_numbers = new int[N + 1];

    // Ініціалізуємо масив чисел за допомогою оптимізованого "Wheel factorization"
    for (int i = 0; i <= N; ++i) {
        h_numbers[i] = 1;
    }
    for (int i = 2; i <= sqrt(N); ++i) {
        if (h_numbers[i] == 1) {
            for (int j = i * i; j <= N; j += i) {
                h_numbers[j] = 0;
            }
        }
    }

    cudaMemcpy(d_numbers, h_numbers, (N + 1) * sizeof(int), cudaMemcpyHostToDevice);

    int blockSize = BLOCK_SIZE;
    int gridSize = (N + blockSize - 1) / blockSize;

    sieveOfEratosthenes << <gridSize, blockSize >> > (d_numbers, N);

    cudaMemcpy(h_numbers, d_numbers, (N + 1) * sizeof(int), cudaMemcpyDeviceToHost);

    clock_t end = clock(); // Кінець таймера
    // Обчислення часу виконання
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

    std::cout << "Searching for prime numbers by the Sieve of Eratosthenes with Wheel
Factorization using NVIDIA CUDA\n";
    std::cout << "Search range from 0 to " << N << "\n";
    std::cout << "Execution time: " << time_spent << " seconds\n";

    int count_of_primes = 0;
    int max_of_primes = 0;
    for (int i = 0; i <= N; ++i) {
        if (h_numbers[i] == 1) {
            ++count_of_primes;
            max_of_primes = i;
        }
    }

    std::cout << "Prime numbers found: " << count_of_primes << "\n";
    std::cout << "Largest of prime numbers: " << max_of_primes << "\n";

    delete[] h_numbers;
    cudaFree(d_numbers);

    return 0;
}
```

Лістинг файлу Eratosfen_Wheel_Linear.cpp пошуку простих чисел з колісною факторизацією з CPU-реалізацією

```
// Функція для пошуку простих чисел за допомогою решета Ератосфена
std::vector<int> sieveOfEratosthenes(int n) {
    std::vector<bool> prime(n + 1, true); // Ініціалізуємо вектор для запам'ятовування
простих чисел
    std::vector<int> primes; // Вектор для зберігання знайдених простих чисел

    // Застосовуємо решето Ератосфена
    for (int p = 2; p * p <= n; ++p) {
        if (prime[p]) {
```

```
            for (int i = p * p; i <= n; i += p) {
                prime[i] = false; // Викреслюємо всі кратні числа
            }
        }
    }

    // Заповнюємо вектор простими числами
    for (int p = 2; p <= n; ++p) {
        if (prime[p]) {
            primes.push_back(p);
        }
    }

    return primes;
}

int main() {
    const int N = 1000000; // Верхня межа пошуку простих чисел

    std::cout << "Searching for prime numbers using the Sieve of Eratosthenes\n";
    std::cout << "Search range from 2 to " << N << "\n";

    clock_t begin = clock(); // Початок вимірювання часу

    // Знаходимо прості числа за допомогою решета Ератосфена
    std::vector<int> primes = sieveOfEratosthenes(N);

    clock_t end = clock(); // Кінець вимірювання часу
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC; // Обчислення часу
виконання

    // Виведення часу виконання
    std::cout << "Execution time: " << time_spent << " seconds\n";
    // Виведення кількості знайдених простих чисел
    std::cout << "Prime numbers found: " << primes.size() << "\n";

    // Виведення найбільшого простого числа
    if (!primes.empty()) {
        std::cout << "Largest prime number: " << primes.back() << "\n";
    }
    else {
        std::cout << "No prime numbers found\n";
    }
    return 0;
}
```

Лістинг файлу AtkinLinear.cpp для пошуку простих чисел решетом Аткіна з CPU-реалізацією

```
// Функція для знаходження простих чисел за допомогою решета Аткіна
void sieveOfAtkin(int* numbers, int n) {
    if (n <= 2) return;

    // Ініціалізуємо масив чисел за допомогою решета Аткіна
    for (int i = 0; i <= n; ++i) {
        numbers[i] = 0;
    }

    for (int x = 1; x <= std::sqrt(n); ++x) {
        for (int y = 1; y <= std::sqrt(n); ++y) {
            int num = 4 * x * x + y * y;
            if (num <= n && (num % 12 == 1 || num % 12 == 5))
                numbers[num] ^= 1;
```

```
            num = 3 * x * x + y * y;
            if (num <= n && num % 12 == 7)
                numbers[num] ^= 1;

            num = 3 * x * x - y * y;
            if (x > y && num <= n && num % 12 == 11)
                numbers[num] ^= 1;
        }
    }
    // Відмітимо числа, кратні квадратам простих чисел
    for (int r = 5; r <= std::sqrt(n); ++r) {
        if (numbers[r] == 1) {
            for (int i = r * r; i <= n; i += r * r) {
                numbers[i] = 0;
            }
        }
    }
    // Відмітимо прості числа
    numbers[2] = numbers[3] = 1;
}

int main() {
    const int N = 1000000;  // Верхня межа пошуку простих чисел
    clock_t begin = clock(); // Початок таймера

    int* numbers = new int[N + 1];

    sieveOfAtkin(numbers, N);

    clock_t end = clock(); // Кінець таймера
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC; // Обчислення часу
    std::cout << "Searching for prime numbers by the Sieve of Atkin\n";
    std::cout << "Search range from 0 to " << N << "\n";
    std::cout << "Execution time: " << time_spent << " seconds\n";

    int count_of_primes = 0;
    int max_of_primes = 0;
    for (int i = 0; i <= N; ++i) {
        if (numbers[i] == 1) {
            ++count_of_primes;
            max_of_primes = i;
        }
    }
    std::cout << "Prime numbers found: " << count_of_primes << "\n";
    std::cout << "Largest of prime numbers: " << max of primes << "\n";
    delete[] numbers;

    return 0;
}
```

Лістинг файлу AtkinCuda.cu пошуку простих чисел решетом Аткіна з CUDA-реалізацією

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#define BLOCK_SIZE 256
// CUDA-ядро для знаходження простих чисел за допомогою решета Аткіна
__global__ void sieveOfAtkin(int* numbers, int n) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    if (tid <= 2 || tid > n) return;
    int x = tid * tid;
```

```
if (x > n) return;
    for (int y = 1; y <= n; ++y) {
        int y_squared = y * y;
        int num = 4 * x + y_squared;

        if (num <= n && (num % 12 == 1 || num % 12 == 5))
            numbers[num] ^= 1;

        num = 3 * x + y_squared;

        if (num <= n && num % 12 == 7)
            numbers[num] ^= 1;

        num = 3 * x - y_squared;

        if (x > y && num <= n && num % 12 == 11)
            numbers[num] ^= 1;
    }
}

int main() {
    const int N = 1000000;  // Верхня межа пошуку простих чисел
    clock_t begin = clock(); // Початок таймера

    int* d_numbers;
    cudaMalloc((void**)&d_numbers, (N + 1) * sizeof(int));
    int* h_numbers = new int[N + 1];

    // Ініціалізуємо масив чисел за допомогою решета Аткіна
    for (int i = 0; i <= N; ++i) {
        h_numbers[i] = 0;
    }

    int blockSize = BLOCK_SIZE;
    int gridSize = (N + blockSize - 1) / blockSize;

    sieveOfAtkin << <gridSize, blockSize >> > (d_numbers, N);

    cudaMemcpy(h_numbers, d_numbers, (N + 1) * sizeof(int), cudaMemcpyDeviceToHost);

    clock_t end = clock(); // Кінець таймера
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC; // Обчислення часу
виконання

    std::cout << "Searching for prime numbers by the Sieve of Atkin using NVIDIA
CUDA\n";
    std::cout << "Search range from 0 to " << N << "\n";
    std::cout << "Execution time: " << time_spent << " seconds\n";

    int count_of_primes = 0;
    int max_of_primes = 0;
    for (int i = 0; i <= N; ++i) {
        if (h_numbers[i] == 1) {
            ++count_of_primes;
            max_of_primes = i;
        }
    }
    std::cout << "Prime numbers found: " << count_of_primes << "\n";
    std::cout << "Largest of prime numbers: " << max_of_primes << "\n";

    delete[] h_numbers;
    cudaFree(d_numbers);
    return 0;
}
```

Лістинг файлу MillerRabinCuda.cu пошуку простих чисел за тестом Мілера-Рабіна з CUDA-реалізацією

```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"
#include <curand.h>
#include <curand_kernel.h>

#include <iostream>
#include <ctime>

// Функція тесту Міллера-Рабіна з локальними змінними d та state
__device__ bool millerRabinTest(unsigned long long n, int k, curandState& state,
unsigned long long d) {
    // Базові випадки
    if (n <= 1 || n == 4)
        return false;
    if (n <= 3)
        return true;

    // Проводимо k ітерацій тесту Міллера-Рабіна
    for (int i = 0; i < k; i++) {
        // Вибираємо випадкове ціле число у діапазоні [2, n-2]
        unsigned long long a = curand(&state) % (n - 3) + 2;

        // Знаходимо a^d % n
        unsigned long long res = 1; // Ініціалізуємо результат

        a = a % n; // Обчислюємо a у модулі n
        while (d > 0) {
            // Якщо y непарне, множимо результат на a
            if (d & 1)
                res = (res * a) % n;
            // y ділимо на 2
            d = d >> 1;
            a = (a * a) % n;
        }

        // Якщо одна з наступних двох умов виконується, то n не є простим
        if (res == 1 || res == n - 1)
            continue;

        // Повторюємо тест для (n-1)/2 разів
        while (d != n - 1) {
            res = (res * res) % n;
            d *= 2;

            if (res == 1)
                return false;
            if (res == n - 1)
                break;
        }

        if (res != n - 1)
            return false;
    }

    // Якщо всі тести пройдені, n має високий шанс бути простим
    return true;
}
```

```
// CUDA-ядро для перевірки чисел на простоту за допомогою тесту Міллера-Рабіна
__global__ void findPrimesInRange(int* primes, int n, int k) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx >= 2 && idx <= n) {
        // Ініціалізуємо генератор випадкових чисел для кожного потоку
        curandState state;
        curand_init(clock() + idx, 0, 0, &state);

        // Копіюємо значення n - 1 у локальну змінну для кожного потоку
        unsigned long long d = n - 1;
        while (d % 2 == 0)
            d /= 2;
        primes[idx] = millerRabinTest(idx, k, state, d) ? 1 : 0;
    }
}

int main() {
    const int N = 100000;  // Задаємо верхню межу пошуку простих чисел
    const int Iteration = 3; // Кількість ітерацій тесту Міллера-Рабіна

    // Запускаємо таймер
    double time_spent = 0.0;
    clock_t begin = clock();

    // Виділяємо пам'ять під масив чисел та копіюємо його на пристрій
    int* d_primes;
    cudaMalloc((void**)&d_primes, (N + 1) * sizeof(int));

    // Задаємо параметри сітки та блоку потоків
    int blockSize = 256;
    int gridSize = (N + blockSize - 1) / blockSize;

    // Запускаємо CUDA-ядро для пошуку простих чисел
    findPrimesInRange << <gridSize, blockSize >> > (d_primes, N, Iteration);

    // Копіюємо результат назад на хост
    int* h_primes = new int[N + 1];
    cudaMemcpy(h_primes, d_primes, (N + 1) * sizeof(int), cudaMemcpyDeviceToHost);

    clock_t end = clock();
    time_spent += (double)(end - begin) / CLOCKS_PER_SEC;

    // Виводимо діапазон пошуку та час виконання
    std::cout << "Searching for prime numbers using Miller-Rabin test with CUDA" <<
std::endl;
    std::cout << "Search range from 0 to " << N << std::endl;
    std::cout << "Execution time: " << time_spent << " seconds" << std::endl;

    // Вивід кількості та максимального простого числа
    int countPrimes = 0;
    int maxPrime = 0;
    for (int i = 0; i <= N; ++i) {
        if (h_primes[i] == 1) {
            countPrimes++;
            maxPrime = i;
        }
    }
    std::cout << "Prime numbers found: " << countPrimes << std::endl;
    std::cout << "Largest prime number: " << maxPrime << std::endl;
    // Звільняємо пам'ять
    delete[] h_primes;
    cudaFree(d_primes);

    return 0;
}
```

Лістинг файлу MillerRabinCuda.cu пошуку простих чисел за тестом Мілера-Рабіна з CPU-реалізацією

```
#include <iostream>
#include <vector>
#include <cmath>
#include <cstdlib>
#include <ctime>

using namespace std;

// Функція для швидкого піднесення до степеня у модулі
long long power(long long x, unsigned long long y, long long p) {
    long long res = 1; // Ініціалізуємо результат

    x = x % p; // Обчислюємо x у модулі p

    while (y > 0) {
        // Якщо y непарне, множимо результат на x
        if (y & 1)
            res = (res * x) % p;

        // y ділимо на 2
        y = y >> 1;
        x = (x * x) % p;
    }
    return res;
}

// Перевірка на простоту за тестом Міллера-Рабіна
bool isPrime(unsigned long long n, int k) {
    // Базові випадки
    if (n <= 1 || n == 4)
        return false;
    if (n <= 3)
        return true;

    // Знаходимо r у виразі n = 2^d * r + 1
    unsigned long long d = n - 1;
    while (d % 2 == 0)
        d /= 2;

    // Проводимо k ітерацій тесту Міллера-Рабіна
    for (int i = 0; i < k; i++) {
        // Вибираємо випадкове ціле число у діапазоні [2, n-2]
        unsigned long long a = 2 + rand() % (n - 4);

        // Знаходимо a^d % n
        unsigned long long x = power(a, d, n);

        // Якщо одна з наступних двох умов виконується, то n не є простим
        if (x == 1 || x == n - 1)
            continue;

        // Повторюємо тест для (n-1)/2 разів
        while (d != n - 1) {
            x = (x * x) % n;
            d *= 2;

            if (x == 1)
                return false;
            if (x == n - 1)
                break;
        }
```

```
            if (x != n - 1)
                return false;
        }
        // Якщо всі тести пройдені, n має високий шанс бути простим
        return true;
}

// Функція для пошуку всіх простих чисел у заданому діапазоні
vector<unsigned long long> findPrimesInRange(unsigned long long end, int k) {
        vector<unsigned long long> primes;
        // Ітеруємося через заданий діапазон
        for (unsigned long long n = 3; n <= end; ++n) {
            if (isPrime(n, k)) {
                primes.push_back(n);
            }
        }
        return primes;
}

int main() {
        const int NumOfPrimes = 1000000000; // Границя діапазону пошуку";
        const int Iteration = 3; // Кількість ітерацій тесту, впливає на точність

        std::cout << "Searching for prime numbers using Miller-Rabin test\n";
        std::cout << "Search range from 3 to " << NumOfPrimes << "\n";

        clock_t begin = clock(); // Початок вимірювання часу
        // Знаходимо всі прості числа у заданому діапазоні
        vector<unsigned long long> primes = findPrimesInRange(NumOfPrimes, Iteration);
        clock_t end = clock(); // Кінець вимірювання часу
        // Обчислення часу виконання
        double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
        // Виводимо кількість знайдених простих чисел
        cout << "Prime numbers found: " << primes.size() << endl;

        // Виводимо останнє знайдене просте число
        cout << "Largest prime number: " << primes.at(primes.size() - 1) << endl;

        // Виведення часу виконання
        std::cout << "Execution time: " << time_spent << " seconds\n";

        return 0;
}
```

Лістинг файлу Lucas-Lehmer_Cuda.cu пошуку простих чисел Мерсена з перевіркою простоти за тестом Люка-Лемера з CUDA-реалізацією

```
#include <iostream>
#include <cmath>
#include <ctime>
#include <cuda_runtime.h>
#include "device_launch_parameters.h"

// Підключення необхідних бібліотек

// Функція для перевірки, чи є число простим
__device__ bool isPrime(int n) {
        if (n <= 1) return false;
        if (n <= 3) return true;
        if (n % 2 == 0 || n % 3 == 0) return false;
        for (int i = 5; i * i <= n; i += 6) {
            if (n % i == 0 || n % (i + 2) == 0) return false;
        }
```

```
    return true;
}

// Функція для виконання тесту Лукаса-Лемера для визначення простоти Мерсеннівих чисел
__device__ bool lucasLehmerTest(int p) {
    if (p == 2) return true;

    long long s = 4;
    long long M = (1LL << p) - 1;

    for (int i = 0; i < p - 2; ++i) {
        s = (s * s - 2) % M;
    }

    return (s == 0);
}

// Ядро CUDA для пошуку простих чисел Мерсенна
__global__ void findMersennePrimes(int N, int* count_of_primes, int* max_of_primes) {
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N) {
        if (isPrime(tid) && lucasLehmerTest(tid)) {
            int index = atomicAdd(count_of_primes, 1); // Атомарне додавання до
лічильника простих чисел
            max_of_primes[index] = tid; // Зберігання простого числа в масиві
        }
    }
}

int main() {
    const int N = 10; // Розмір пошукового діапазону Мерсенна чисел

    std::cout << "Testing (2^p-1) Mersenne numbers for primality using the Lucas-Lehmer
test with NVIDIA CUDA\n";
    std::cout << "Search range: p = 2.. " << N << "\n";

    clock_t begin = clock();
    int* count_of_primes_d;
    int* max_of_primes_d;
    int count_of_primes = 0;
    int max_of_primes[N]; // Масив для збереження простих чисел

    cudaMalloc((void**)&count_of_primes_d, sizeof(int));
    cudaMalloc((void**)&max_of_primes_d, N * sizeof(int));
    cudaMemcpy(count_of_primes_d, &count_of_primes, sizeof(int),
               cudaMemcpyHostToDevice);

    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;

    findMersennePrimes << <blocksPerGrid, threadsPerBlock >> > (N, count_of_primes_d,
               max_of_primes_d);

    cudaMemcpy(&count_of_primes, count_of_primes_d, sizeof(int),
cudaMemcpyDeviceToHost);
    cudaMemcpy(max_of_primes, max_of_primes_d, count_of_primes * sizeof(int),
cudaMemcpyDeviceToHost);
    clock_t end = clock();
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;
    std::cout << "\nExecution time: " << time_spent << " seconds\n";

    std::cout << "Prime numbers found: " << count_of_primes << "\n";
    // Індексація останнього знайденого простого числа
    std::cout << "Largest of prime numbers: " << "2^" << max_of_primes[count_of_primes -
1] << "-1\n";
```

```
    cudaFree(count_of_primes_d);
    cudaFree(max_of_primes_d);
    return 0;
}
```

Лістинг файлу LucasLehmer.cpp пошуку простих чисел Мерсена з перевіркою простоти за тестом Люка-Лемера з CPU-реалізацією

```
#include <iostream>
#include <cmath>
#include <ctime>

// Функція для перевірки простоти числа Мерсена
bool isPrime(int n) {
    if (n <= 1) return false;
    if (n <= 3) return true;
    if (n % 2 == 0 || n % 3 == 0) return false;
    for (int i = 5; i * i <= n; i += 6) {
        if (n % i == 0 || n % (i + 2) == 0) return false;
    }
    return true;
}

// Функція для перевірки простоти числа за допомогою тесту Люка-Лемера
bool lucasLehmerTest(int p) {
    if (p == 2) return true; // 2^2 - 1 = 3, яке є простим

    long long s = 4;
    long long M = (1LL << p) - 1; // просте число Мерсена (2^p - 1)

    for (int i = 0; i < p - 2; ++i) {
        s = (s * s - 2) % M;
    }
    return (s == 0);
}

int main() {
    const int N = 500000;  // Максимальне p для тесту Люка-Лемера
    std::cout << "Searching for Mersenne primes (2^p - 1) using Lucas-Lehmer test\n";
    std::cout << "Search range: p = 2.. " << N << "\n";

    clock_t begin = clock(); // Початок таймера
    int count_of_primes = 0;
    int max_of_primes = 0;

    for (int p = 2; p <= N; ++p) {
        if (isPrime(p) && lucasLehmerTest(p)) {
            ++count_of_primes;
            max_of_primes = p;
        }
    }
    clock_t end = clock(); // Кінець таймера
    // Обчислення часу виконання
    double time_spent = (double)(end - begin) / CLOCKS_PER_SEC;

    std::cout << "Prime numbers found: " << count_of_primes << "\n";
    std::cout << "Largest of prime numbers: " << "2^" << max_of_primes << "-1\n";
    std::cout << "\nExecution time: " << time_spent << " seconds\n";
    return 0;}
```