МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Західноукраїнський національний університет
Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління

РОМАНЮК Ростислав Сергійович

Програмний модуль паралельної обробки сенсорних потоків IoT на базі технології CUDA / A software module for parallel processing of IoT sensor flows based on CUDA

Спеціальність 122 – Комп'ютерні науки
Освітньо-професійна програма – Комп'ютерні науки

Кваліфікаційна робота

Виконав студент групи КН-41
Р. С. Романюк

Науковий керівник:
к.т.н., доцент, О.Р. Осолінський

Кваліфікаційну роботу допущено до захисту
«___»_____ 2026 р.

Завідувач кафедри
_____Н.В. Дзюбановська

Тернопіль – 2026

Факультет комп'ютерних інформаційних технологій
Кафедра інформаційно-обчислювальних систем і управління
Освітній ступінь «бакалавр»
спеціальність 122 – Комп'ютерні науки
освітньо-професійна програма – Комп'ютерні науки

ЗАТВЕРДЖУЮ
В.о. завідувача кафедри
_____ Н.В. Дзюбановська
«____» _____20__ р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ
РОМАНЮКУ Ростислав Сергійович
(прізвище, ім'я, по батькові)

1. Тема кваліфікаційної роботи:
Програмний модуль паралельної обробки сенсорних потоків IoT на базі технології CUDA / A software module for parallel processing of IoT sensor flows based on CUDA

керівник роботи О.Р. Осолінський, к.т.н., доцент
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)
затверджений наказом по університету від 01 грудня 2025 року № 894.
2. Строк подання студентом закінченої кваліфікаційної роботи 25 травня 2026 р.
3. Вихідні дані до кваліфікаційної роботи: завдання на кваліфікаційну роботу студента, наукові статті, технічна література.

4. Основні питання, які потрібно розробити:
- провести аналіз формування сенсорних потоків даних в IoT-системах;
- провести огляд методів і засобів паралельної обробки даних;
- проаналізувати існуючі програмні рішення та платформи;
- сформувати постановку задачі;
- розробити архітектуру програмного модуля паралельної обробки;
- розробити алгоритми функціонування компонент модуля;
- розробити модель взаємодії CPU та GPU при обробці сенсорних потоків;
- провести вибір програмних і технічних засобів реалізації;
- провести програмну реалізацію модуля обробки сенсорних даних;
- провести дослідження ефективності програмного модуля.

5. Перелік графічного матеріалу в роботі:
- розширена архітектура програмного модуля паралельної обробки сенсорних потоків IoT;
- діаграма послідовності обробки пакета сенсорних даних у взаємодії CPU та GPU.

6. Консультанти розділів кваліфікаційної роботи

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		Завдання видав	Завдання прийняв

7. Дата видачі завдання 01 грудня 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів кваліфікаційної роботи	Примітка
1	Огляд літературних джерел відповідно до предметної області та складання плану роботи	до 01.01.2026 р.	
2	Написання 1 розділу кваліфікаційної роботи	до 01.02.2026 р.	
3	Написання 2 розділу кваліфікаційної роботи	до 15.03.2026 р.	
4	Написання 3 розділу кваліфікаційної роботи	до 01.05.2026 р.	
5	Представлення попереднього варіанту кваліфікаційної роботи, перевірка та внесення змін керівником	до 15.05.2026 р.	
6	Опрацювання зауважень та представлення завершеного варіанту кваліфікаційної роботи. Підготовка супроводжуючих документів	до 25.05.2026 р.	
7	Перевірка кваліфікаційної роботи на оригінальність тексту	до 30.05.2026 р.	
8	Оформлення кваліфікаційної роботи та отримання допуску до захисту	до 10.06.2026 р.	
9	Подання кваліфікаційної роботи до захисту на засіданні атестаційної комісії	до 10.06.2026 р.	

Студент_____ Р. С. Романюк
(підпис) (прізвище та ініціали)

Керівник кваліфікаційної роботи_____ О.Р. Осолінський
(підпис) (прізвище та ініціали)

АНОТАЦІЯ

Пояснювальна записка до кваліфікаційної роботи: 59 с., 17 рис., 47 джерел, 4 додатки.

Метою роботи є розробка програмного модуля паралельної обробки сенсорних потоків IoT на базі технології CUDA.

В роботі застосовано методи системного аналізу, логічного структурування, модульного проєктування, програмування мовами C++ і Python, технологію CUDA, засоби організації взаємодії CPU та GPU.

Запропоновано архітектуру програмного модуля паралельної обробки сенсорних потоків, яка включає джерела даних, модуль приймання потоку, буфер пам’яті, CPU-частину керування, GPU-ядра CUDA, модуль формування результатів та інтерфейс взаємодії із зовнішніми компонентами системи.

Розроблено алгоритми функціонування основних компонентів модуля, що охоплюють приймання сенсорних повідомлень, буферизацію та підготовку пакета, передавання даних між CPU і GPU, паралельну обробку на графічному процесорі, повернення результатів і постобробку.

Реалізовано програмний модуль, який підтримує режими обробки на CPU та GPU, виконує нормалізацію даних, порогову перевірку, формування статистики та запис результатів. Також розроблено графічний інтерфейс, який забезпечує генерацію вхідних даних, вибір режиму обробки, запуск обчислень і перегляд результатів.

Проведено тестування програмного модуля на згенерованих наборах сенсорних даних. Результати підтвердили коректність роботи в режимах CPU і GPU та показали доцільність використання графічного процесора для великих обсягів даних.

Ключові слова: Інтернет речей, сенсорні потоки, паралельна обробка, CUDA, GPU, CPU, програмний модуль, C++, Python, графічний інтерфейс.

ANNOTATION

Explanatory note to the qualification thesis: 59 pages, 17 figures, 47 references, 4 appendices.

The aim of the work is to develop a software module for parallel processing of IoT sensor streams based on CUDA technology.

The work applies methods of systems analysis, logical structuring, modular design, programming in C++ and Python, CUDA technology, and tools for organizing CPU–GPU interaction.

An architecture of a software module for parallel processing of sensor streams is proposed, which includes data sources, a stream reception module, a memory buffer, a CPU control unit, CUDA GPU kernels, a result generation module, and an interface for interaction with external system components.

Algorithms for the functioning of the main module components have been developed, covering sensor message reception, buffering and packet preparation, data transfer between CPU and GPU, parallel processing on the graphics processor, return of results, and post-processing.

A software module has been implemented that supports CPU and GPU processing modes, performs data normalization, threshold checking, statistical calculation, and result recording. A graphical user interface has also been developed, providing input data generation, processing mode selection, computation launch, and result viewing.

The software module was tested on generated sets of sensor data. The results confirmed the correctness of operation in both CPU and GPU modes and demonstrated the feasibility of using a graphics processor for large data volumes.

Keywords: Internet of Things, sensor streams, parallel processing, CUDA, GPU, CPU, software module, C++, Python, graphical user interface.

ЗМІСТ

Вступ 7
1 Стан та аналіз предметної області 9
1.1 Особливості сенсорних потоків даних в IoT-системах 9
1.2 Огляд методів і засобів паралельної обробки даних 12
1.3 Аналіз існуючих програмних рішень та платформ 15
1.4 Постановка задачі 18
2 Алгоритмічне та інформаційне забезпечення програмного модуля 20
2.1 Архітектура програмного модуля паралельної обробки 20
2.2 Алгоритми функціонування компонент модуля 23
2.3 Модель взаємодії CPU та GPU при обробці сенсорних потоків 27
3 Програмно-технологічне забезпечення модуля 32
3.1 Вибір програмних і технічних засобів реалізації 32
3.2 Програмна реалізація модуля обробки сенсорних даних 34
3.3 Дослідження ефективності програмного модуля 38
Висновки 41
Список використаних джерел 43
Додаток А Розширена архітектура програмного модуля паралельної обробки сенсорних потоків IoT 48
Додаток Б Діаграма послідовності обробки пакета сенсорних даних у взаємодії CPU та GPU 49
Додаток В Код програмно апаратного модуля 50
Додаток Г Апробація отриманих результатів 55

ВСТУП

Сучасний розвиток інформаційних технологій тісно пов’язаний з поширенням систем Інтернету речей, які забезпечують автоматизований збір, передавання, обробку та аналіз даних від фізичних об’єктів. IoT-рішення застосовуються в промисловості, міській інфраструктурі, транспорті, сільському господарстві, енергетиці, екологічному моніторингу та багатьох інших сферах, де необхідно постійно контролювати стан об’єктів, середовища або процесів. Основою таких систем є сенсорні пристрої, які формують потоки даних у вигляді послідовностей вимірювань температури, вологості, тиску, освітленості, прискорення, концентрації газів та інших параметрів.

Особливістю сенсорних потоків у IoT-системах є те, що вони можуть надходити безперервно, з різною інтенсивністю та від великої кількості пристроїв. Такі потоки мають часову природу, містять ідентифікатори пристроїв, часові мітки, службові поля та набори виміряних значень. У практичних умовах вони є нерівномірними, різнорідними за структурою та чутливими до затримок обробки. Тому для сучасних IoT-систем важливими є швидке приймання даних, їх попередня фільтрація, нормалізація, агрегація, виявлення некоректних значень і підготовка результатів для подальшого аналізу або візуалізації.

Із збільшенням кількості сенсорів, частоти надходження повідомлень і складності обчислень традиційна послідовна обробка на центральному процесорі не завжди забезпечує достатню продуктивність. Тому актуальним є застосування технології CUDA, яка дає змогу організувати масово-паралельну обробку даних на GPU та підвищити швидкодію виконання обчислювально інтенсивних операцій.

У зв’язку з цим виникає задача розробки програмного модуля, який забезпечуватиме паралельну обробку потоків сенсорних даних IoT з використанням технології CUDA та буде орієнтований на виконання масових однотипних операцій над великими наборами вхідних даних у потоковому режимі.

Метою кваліфікаційної роботи є розробка програмного модуля паралельної обробки сенсорних потоків IoT на базі технології CUDA.

Завдання кваліфікаційної роботи полягають у послідовному виконанні таких етапів:

– розробити архітектуру програмного модуля паралельної обробки потоків сенсорних даних IoT на базі технології CUDA;

– розробити алгоритми функціонування ключових компонентів програмного модуля;

– розробити інформаційне забезпечення програмного модуля, структури вхідних і вихідних даних та організацію їх передавання між CPU і GPU;

– провести вибір програмних і апаратних засобів реалізації;

– реалізувати ключові модулі паралельної обробки даних з використанням технології CUDA;

– реалізувати інтерфейс керування та взаємодії з користувачем;

– провести тестування працездатності програмного модуля та оцінити його продуктивність.

Об’єктом дослідження є процеси приймання, буферизації, передавання та обробки сенсорних потоків даних в IoT-системах.

Предметом дослідження є моделі, алгоритми та програмно-технологічні засоби паралельної обробки сенсорних потоків із використанням CPU/GPU-архітектури та технології CUDA.

Результати кваліфікаційної роботи апробовані та опубліковані у матеріалах IX Всеукраїнської студентської наукової конференції «Науковий простір: аналіз, сучасний стан, тренди та перспективи» (м. Київ, Україна).

Кваліфікаційна робота складається із вступу, трьох розділів, висновків, списку використаних джерел та додатків.

1 СТАН ТА АНАЛІЗ ПРЕДМЕТНОЇ ОБЛАСТІ

1.1 Особливості сенсорних потоків даних в IoT-системах

Сенсорні потоки даних в IoT-системах являють собою безперервні або подієво-ініційовані послідовності вимірювань, які формуються фізичними сенсорами під час контролю стану об’єкта, середовища або процесу. Такі потоки є основою функціонування систем Інтернету речей, тому що вони забезпечують перехід від вимірювання параметрів фізичного світу до їх подальшої передачі, обробки, аналізу та прийняття рішень. В роботах [1, 2], присвячених обробці сенсорних даних IoT, показується, що сенсорна мережа крім того, що фіксує показники зовнішнього середовища ще і перетворює первинні вимірювання у змістовну інформацію, придатну для керування, прогнозування або сповіщення.

По суті та технічній характеристиці сенсорні потоки найчастіше мають часовий характер, тобто є часовими рядами, які генеруються періодично або при настанні певної події. В дослідженні [3] представлено, що частота дискретизації може змінюватися від одного вимірювання на добу до сотень вимірювань за секунду, а кількість джерел даних — від сотень до мільйонів сенсорів. В результаті реальна інтенсивність вхідних потоків може змінюватися в дуже широких межах — від дуже низьких значень до 10^5 повідомлень за секунду, тоді як в практичних тестових навантаженнях для IoT-платформ розглядалися пікові режими 500–10000 повідомлень за секунду. Це означає, що сенсорний потік в IoT не є статичним набором записів, а динамічним безперервним середовищем даних зі змінною інтенсивністю надходження [4].

Типи даних, які генерують IoT-пристрої, також є різнорідними. До типових належать числові значення температури, тиску, вологості, рівня концентрації газу, прискорення, кутової швидкості, освітленості та інших параметрів.

Окремим класом йдуть RFID-дані, інфрачервоні сигнали, зображення та відео, а також складені повідомлення, що містять часову мітку, ідентифікатор пристрою й набір полів спостереження. Крім того представлення таких даних може бути бінарним, булевим, числовим, безперервним або у вигляді набору ознак. Така

різноманітність формує одну з ключових рис сенсорних потоків — гетерогенність, яка ускладнює уніфікований збір, синхронізацію та подальший аналіз [5, 6].

Важливою особливістю сенсорних потоків є те, що вони формуються в умовах обмежених ресурсів. Більшість IoT-пристроїв мають невеликі обчислювальні можливості, обмежений обсяг пам’яті, низький запас енергії та невисоку пропускну здатність каналу зв’язку. У зв’язку з цим в архітектурі IoT-систем значну роль відіграють механізми очищення, агрегування, виявлення викидів, відновлення пропущених значень та злиття даних з кількох джерел.

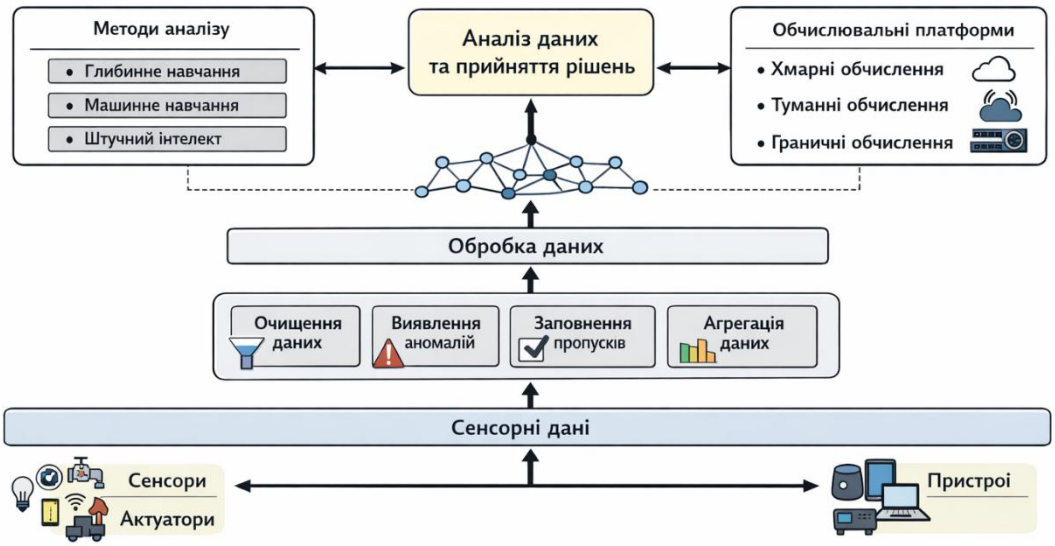


Рисунок 1.1 – Узагальнена структура обробки сенсорних потоків даних в IoT-системі

Такі процедури дозволяють зменшити обсяг передаваних повідомлень і підвищити інформативність потоку ще до етапу глибшої аналітики [7, 8].

Для сенсорних потоків в IoT характерні підвищені вимоги до швидкодії та затримки обробки. Це пояснюється тим, що багато застосувань працюють по замкненому циклу, тобто спостереження, аналіз, а потім дія, де рішення повинно бути вироблене в реальному часі або в режимі, наближеному до реального часу.

Тому для IoT-середовищ критично важливими є низька латентність, висока пропускна здатність і можливість безперервної потокової обробки. Центральна хмарна обробка не завжди здатна задовольнити ці вимоги через додаткові затримки передавання, тому все частіше застосовуються edge- і fog-підходи, де обчислення переносяться ближче до джерела даних. Такий підхід зменшує затримку, скорочує навантаження на мережу й підвищує адаптивність системи [9-11].

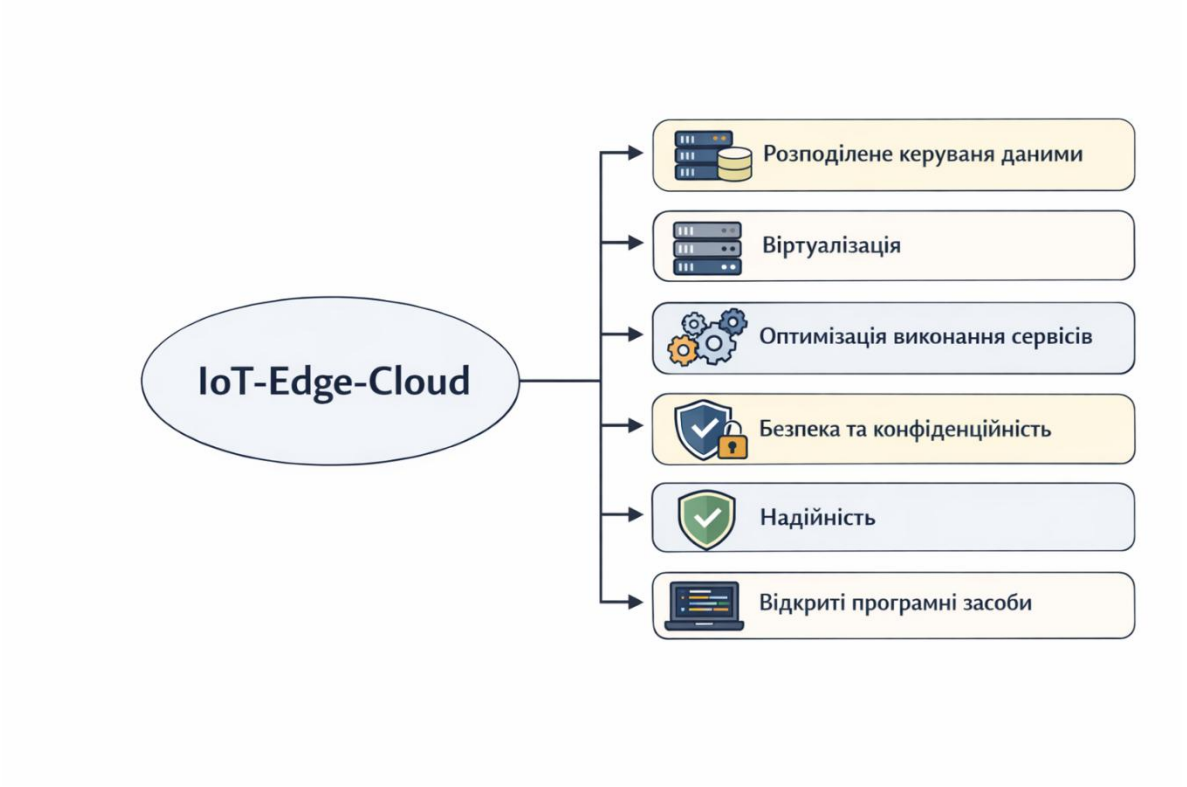


Рисунок 1.2 – Основні вимоги та проблеми обробки сенсорних потоків у IoT

Ще однією важливою вимогою є масштабованість. Зі збільшенням кількості підключених пристроїв різко зростають обсяги даних, навантаження на канали передавання та вимоги до засобів зберігання та обробки. В роботах [12, 13], які присвячуються IoT-edge-cloud, масштабованість розглядається разом з гетерогенністю, оптимізацією ресурсів, розподіленим керуванням і забезпеченням надійності. Тому сучасна IoT-система повинна приймати безперервні потоки від великої кількості вузлів та підтримувати стійку роботу за змінних навантажень, нерівномірної частоти надходження повідомлень та можливих відмов окремих

компонентів. Без цього неможливо забезпечити безперервність аналітичного контуру і належну якість сервісу.

Отже сенсорні потоки даних в IoT-системах характеризуються часовою природою, різнорідністю форматів, обмеженнями ресурсів на рівні пристроїв, нерівномірністю інтенсивності надходження та високими вимогами до оперативності обробки. Ці властивості зумовлюють необхідність використання потокових архітектур, локальної попередньої обробки, розподілених моделей обчислень і механізмів масштабування. В такому руслі технологія CUDA є доцільною не тільки з погляду підвищення швидкодії, але й для забезпечення безперервної обробки великих і різнорідних потоків даних у реальному часі [14-16].

1.2 Огляд методів і засобів паралельної обробки даних

Паралельна обробка даних є одним з базових підходів до підвищення продуктивності обчислювальних систем, коли обсяг вхідної інформації зростає, а вимоги до часу реакції стають жорсткішими. В загальному випадку засоби паралельної обробки можна поділити на CPU та GPU-орієнтовані. Підхід на основі центрального процесора ґрунтується на використанні кількох ядер, спільної пам’яті та багатопотокового виконання, тоді як графічний процесор орієнтований на одночасне виконання великої кількості однотипних операцій над великими масивами даних [17, 18]. В задачах, де обробка зводиться до повторюваних обчислень над незалежними елементами потоку[19, 20], GPU-підхід демонструє вищу ефективність, ніж традиційне послідовне або багатоядерне CPU-виконання.

Для CPU-систем найбільш поширеним засобом є багатопотокова обробка, на основі OpenMP. В такій моделі паралелізм досягається за рахунок розбиття циклів, окремих задач або фрагментів алгоритму між потоками в межах спільної пам’яті. Перевагою цього підходу є відносна простота розробки, зручність відлагодження та висока продуктивність для алгоритмів з помірним рівнем паралелізму іскладнішою логікою керування [21]. Водночас масштабування CPU-рішень

обмежується кількістю ядер, конкуренцією за спільні ресурси пам’яті та зростанням накладних витрат на синхронізацію [22]. Тому при переході до великих потоків даних, де потрібно виконувати тисячі або мільйони однотипних операцій, багатопотокового підходу часто виявляється недостатньо [23].

Окреме місце серед CPU-орієнтованих підходів займає модель SIMD/SPMD. SIMD передбачає виконання однієї інструкції над багатьма елементами даних, а SPMD — виконання однієї програми багатьма незалежними екземплярами над різними даними.

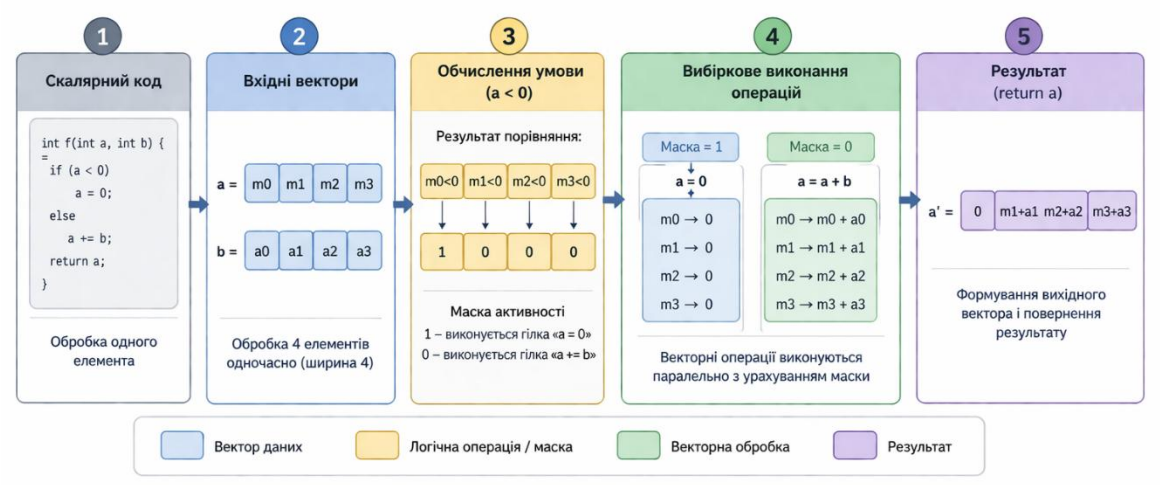


Рисунок 1.3 – Реалізація моделі SPMD на SIMD-векторних засобах центрального процесора

В роботі [24] про компілятор ispc показано, що модель SPMD може ефективно відображатися на SIMD-векторні блоки сучасних процесорів. Це дозволяє використовувати звичну програмну логіку, але водночас задіювати векторні обчислення CPU. Такий підхід є корисним для задач, де потрібно поєднати гнучкість програмування та підвищення продуктивності, але його можливості все одно поступаються масово-паралельним GPU-архітектурам коли йде робота з великими однорідними потоками [25, 26].

GPU-підхід з застосуванням CUDA, орієнтований на масово-паралельну обробку, коли дані можуть бути розподілені між великою кількістю потоків. На відміну від CPU, графічний процесор має значно більше обчислювальних ядер і

краще пристосований до виконання однотипних операцій над великими блоками інформації. Тому CUDA можна використовувати в тих випадках, коли вхідний потік складається з великої кількості незалежних елементів, а основна частина часу витрачається на обчислення, а не на складне логічне керування [27, 28]. В роботі [28] також показано, що платформа CUDA спеціально орієнтована на побудову систем обробки великих масивів даних і потребує врахування моделі пам'яті та особливостей програмування під масово-паралельні архітектури.

В сучасних дослідженнях показано, що для деяких задач найкращі результати дає не чисто GPU-підхід, а гібридна схема CPU+GPU.

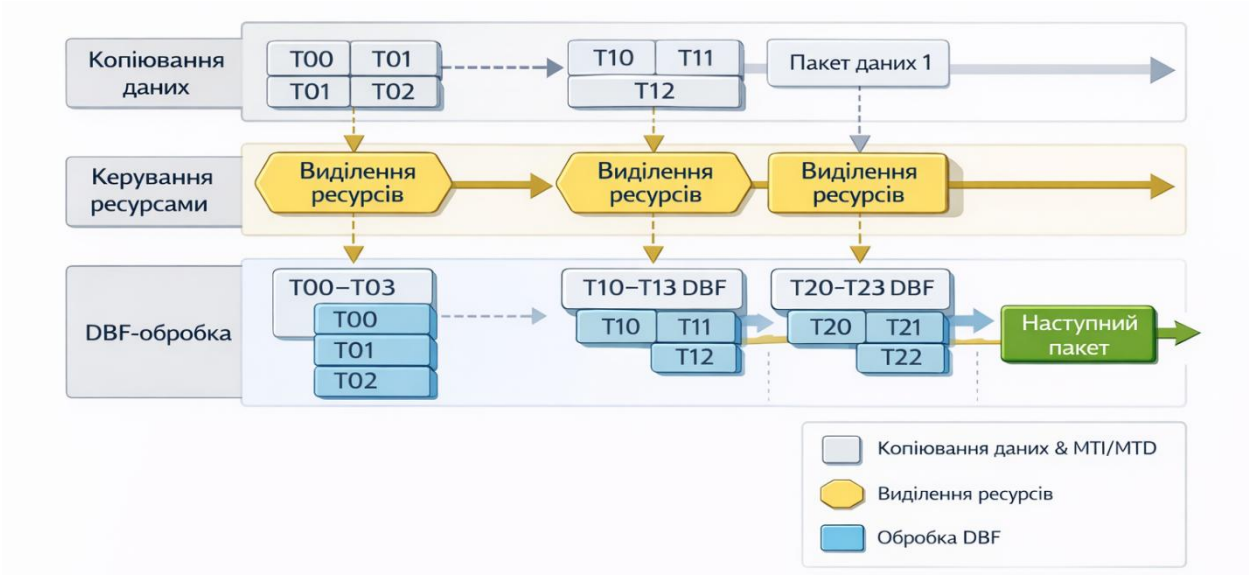


Рисунок 1.4 – Потокова схема гібридної паралельної обробки даних з використанням OpenMP та CUDA

Наприклад, CPU може ефективніше виконувати логічно складні або етапи керування, а GPU буде обчислювати масові фрагменти. В роботі [29], присвяченій гібридному прискоренню міліметрової візуалізації, показано, що саме перенесення логічно складної частини на CPU і обчислювально інтенсивної — на GPU дає вищу швидкодію, ніж використання лише GPU. Подібний підхід наведено в роботі про MIMO-радар [30], де поєднання OpenMP на CPU та CUDA на GPU забезпечило прискорення порівняно з послідовним CPU-виконанням, а також покращення відносно тільки GPU варіанту.

Огляд сучасних методів і засобів паралельної обробки даних показує, що CPU-підходи є доцільними для задач із складнішим керуванням, помірним рівнем паралелізму та використанням спільної пам’яті, тоді як GPU-орієнтовані засоби, такі як CUDA, є більш кращими для обробки великих потоків однотипних даних. Для потоку сенсорних даних IoT це важливо, оскільки такі потоки є масовими, безперервними і часто складаються з незалежних елементарних операцій над наборами вимірювань.

1.3 Аналіз існуючих програмних рішень та платформ

Для задач обробки IoT-потоків в реальному часі сьогодні застосовують кілька груп програмних засобів: системи потокової обробки даних, edge та IoT-платформи, GPU-орієнтовані середовища та бібліотеки для прискорення обчислень на базі CUDA. Кожна з цих груп вирішує свою частину задачі, однак жодна з них у готовому вигляді не є універсальним рішенням для високочастотної обробки великих потоків сенсорних даних. В сучасних оглядах[31] відзначається, що для потокових часових рядів в реальному часі важливими є не лише низька затримка, а й здатність системи безперервно приймати дані, виконувати аналіз, масштабуватися та підтримувати подальше застосування алгоритмів прогнозування або виявлення аномалій.

Серед систем потокової обробки найбільш поширеними є Apache Spark, Apache Flink, Apache Storm, Apache Samza та Amazon Kinesis. Hadoop розглядається переважно як історично важливе рішення для пакетної обробки, але не як повноцінний інструмент для роботи з потоками в реальному часі. Spark підтримує потокову обробку через мікропакети, використовує RDD та має добру інтеграцію з бібліотеками аналітики й машинного навчання, [32]. Flink і Storm [31] реалізують безпосередню потокову обробку, причому Flink дає змогу враховувати фактичний час виникнення подій і коректно працювати з запізнілими даними, що є важливим для роботи із затриманими подіями. В роботі [32] відзначено, що Spark є швидшим за Hadoop завдяки обробці в пам’яті та підтримує потокову обробку в реальному

часі, тоді як Flink орієнтований на роботу зі станом потоку і допускає відкат і повтор обчислень.

Разом з тим наявні потокові платформи мають обмеження. В порівняльному огляді [31] показано, що Spark, хоч і є дуже популярним, не завжди є найкращим вибором для сценаріїв, де потрібні мінімальна затримка і найвищий темп обробки; в проведених експериментах він демонстрував вищі вимоги до оперативної пам'яті та гірші показники latency, ніж Flink. Storm забезпечує низьку затримку, але його масштабування для складніших навантажень є обмеженим, а налагодження і супровід можуть бути складними. Kinesis спрощує збір і аналіз потоків у режимі реального часу, але в роботі [32] показується, що для практичної задачі обробки потокових даних ця платформа вимагає ручної оптимізації, є дорожчою за альтернативи і тісно пов'язує рішення з екосистемою AWS. Отже, готові DSPS-платформи добре підходять для побудови загального аналітичного контуру, але не завжди оптимальні для спеціалізованої обробки IoT-потоків з жорсткими часовими вимогами.

Окрему групу становлять edge та IoT-платформи. В огляді систем периферійних обчислень [30] показано, що такі рішення розміщують обчислювальні, мережеві та сховищні ресурси ближче до джерела даних і тим самим зменшують час реакції. Наведені приклади Cloudlet, CloudPath, ParaDrop, Firework, FocusStack, EdgeX Foundry, Azure IoT Edge та інших платформ. Їхня сильна сторона полягає в тому, що вони дозволяють попередньо фільтрувати, агрегувати й обробляти дані без повної передачі потоку до хмари. Проте в роботі наголошено, що для цього класу систем немає чіткої стандартизації, а серед відкритих проблем залишаються мобільність, багатокористувацька підтримка, захист приватності, енергоефективність і підтримка великих моделей штучного інтелекту на краю мережі [33]. Для задачі обробки сенсорних потоків це означає, що edge-платформа сама по собі ще не розв'язує проблему високопродуктивної аналітики, а лише задає більш придатне середовище виконання.

Рішення які базуються на графічних процесорах розглядаються як відповідь на обчислювальну складність потокової аналітики. В роботі [34] показується, що

графічні процесори добре підходять для прискорення навчання та інференсу, а також для виконання обчислювально інтенсивних задач в режимі, близькому до реального часу. Водночас зазначається, що централізоване виконання глибоких моделей в хмарі спричиняє більшу затримку, підвищує енергетичні та фінансові витрати, а для вбудованих та edge-сценаріїв важливими стають компроміси між продуктивністю, точністю та споживанням ресурсів. В роботі [35] також показано, що навіть сучасні DSPS часто не повністю використовують можливості апаратури, тому що для них характерні накладні витрати на передавання даних між функціональними модулями системи, проблеми з обробкою невпорядкованих потоків, збереженням стану і масштабуванням. Це критично для IoT, де потоки є безперервними, нерівномірними та чутливими до затримки.

Бібліотеки CUDA дають ще нижчий рівень доступу до продуктивності GPU. В роботі [36] наведено, що стек CUDA-X охоплює математичні, комунікаційні, мультимедійні та deep learning бібліотеки, зокрема cuBLAS, cuFFT, cuRAND, cuSPARSE, cuTENSOR, cuSOLVER і cuDNN. Перевага такого підходу полягає у високій ефективності та можливості використання вже оптимізованих примітивів. Але також в роботі представлено, що в гетерогенних системах вузьким місцем залишаються передавання даних між host і device, а бібліотечний рівень, хоч і найпростіший для застосування, не завжди забезпечує найкращу продуктивність для конкретної прикладної задачі. Це означає, що для обробки IoT-потоків в реальному часі сам факт наявності CUDA-бібліотек ще не гарантує оптимального рішення: потрібно спеціально організовувати структуру даних, керування пам’яттю та конвеєр обробки.

Отже, існуючі програмні рішення та платформи дають корисний інструментарій для побудови систем потокової аналітики, але кожен клас засобів має свої межі. DSPS-платформи є універсальними, але мають накладні витрати і не завжди оптимально працюють при жорстких вимогах до latency. Edge-платформи зменшують відстань до джерела даних, але не усувають проблеми стандартизації, енергоефективності та продуктивності. GPU-орієнтовані рішення і CUDA-

бібліотеки забезпечують високу швидкодію, але вимагають детального врахування передачі даних, пам’яті та прикладної специфіки.

1.4 Постановка задачі

В результаті проведеного аналізу предметної області та існуючих програмних рішень показано, що сучасні IoT-системи формують значні обсяги сенсорних даних, які надходять безперервно, нерівномірно та потребують оперативної обробки. В простих системах така обробка часто виконується центральним процесором, але зі збільшенням кількості сенсорів, частоти надходження повідомлень і складності обчислень продуктивності CPU-підходу стає недостатньо. Це призводить до збільшення затримки, накопичення черг повідомлень, втрати актуальності результатів обробки та зниження загальної ефективності системи. Це проявляється у випадках, коли потрібно одночасно виконувати фільтрацію, нормалізацію, агрегацію, перетворення форматів даних та підготовку інформації для подальшого аналізу.

Тому виникає задача розробки програмного модуля, який забезпечуватиме паралельну обробку потоків сенсорних даних IoT з використанням технології CUDA. Модуль повинен бути орієнтований на виконання масових однотипних операцій над великими наборами вхідних даних, що надходять від сенсорних пристроїв в режимі потоку. Основною ідеєю є перенесення найбільш обчислювально навантажених етапів на графічний процесор, що дасть змогу скоротити час обробки та підвищити пропускну здатність системи.

Модуль повинен приймати вхідний потік сенсорних повідомлень, виконувати їх буферизацію, підготовку до обробки, передачу даних до GPU, паралельне виконання визначених операцій та формування вихідного результату в придатному для подальшого використання вигляді. Вхідними даними для модуля є потоки сенсорних значень, які можуть містити часові мітки, ідентифікатори пристроїв, виміряні параметри та службову інформацію. Вихідними даними є

оброблені масиви значень, агреговані результати або підготовлені набори даних для подальшого аналізу, візуалізації чи передавання іншим компонентам системи.

Під час розробки модуля необхідно врахувати кілька груп вимог, а саме модуль має забезпечувати високу швидкодію при обробці великих потоків даних. Крім цього, він повинен підтримувати безперервний режим роботи, бути придатним до масштабування при зростанні інтенсивності потоку, а також забезпечувати коректність результатів обробки. Важливою вимогою є ефективне використання пам’яті CPU та GPU, томущо надлишкові витрати на копіювання даних між host і device можуть зменшити очікуваний виграш від паралелізації. Також модуль повинен мати зрозумілу внутрішню структуру, що дозволить надалі розширювати його функціональні можливості або адаптувати до інших типів потокових даних.

Для досягнення поставленої мети необхідно виконати наступні завдання:

1. Розробити архітектуру програмного модуля паралельної обробки потоків сенсорних даних IoT на базі технології CUDA.
2. Розробити алгоритми функціонування ключових компонентів програмного модуля.
3. Розробити інформаційне забезпечення програмного модуля, структури вхідних і вихідних даних та організацію їх передавання між CPU і GPU.
4. Провести вибір програмних і апаратних засобів реалізації.
5. Реалізувати ключові модулі паралельної обробки даних з використанням технології CUDA.
6. Реалізувати інтерфейс керування та взаємодії з користувачем.
7. Провести тестування працездатності програмного модуля та оцінити його продуктивність.

2 АЛГОРИТМІЧНЕ ТА ІНФОРМАЦІЙНЕ ЗАБЕЗПЕЧЕННЯ ПРОГРАМНОГО МОДУЛЯ

2.1 Архітектура програмного модуля паралельної обробки

З врахуванням проведеного аналізу та постановки завдання архітектуру програмного модуля паралельної обробки потоків сенсорних даних можна будувати за модульним принципом. Такий підхід дозволяє відокремити етап приймання потоку, етап попередньої підготовки даних, етап масово-паралельної обробки на GPU, а також етап формування і передавання результатів. В загальному вигляді модуль взаємодіє з джерелами даних, внутрішнім буфером пам’яті, CPU-частиною керування, GPU-ядрами CUDA та зовнішніми компонентами, які споживають результати.

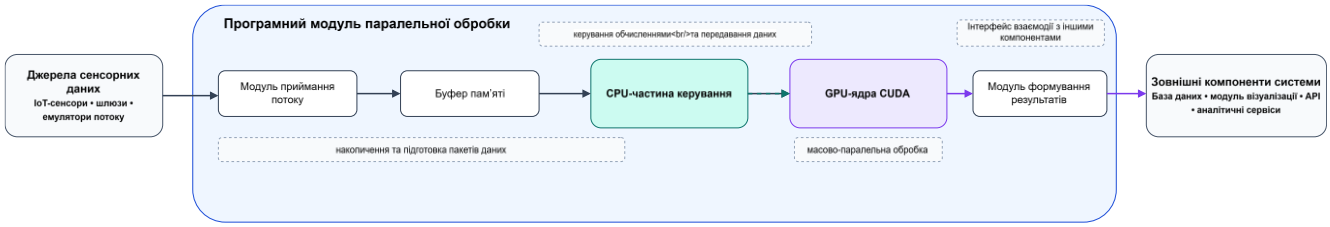


Рисунок 2.1 – Загальна структурна схема програмного модуля паралельної обробки потоків сенсорних даних

Ця логіка узгоджується і з підходами периферійних обчислень, де частина операцій виконується ближче до джерела даних для зменшення затримки, зменшення мережевого навантаження та локальної попередньої обробки потоку.

Першим елементом архітектури є джерела даних. Ними можуть виступати IoT-сенсори, контролери, шлюзи або програмні емулятори потоку. Дані надходять у вигляді послідовності повідомлень, які містять ідентифікатор пристрою, часову мітку, виміряні значення та службові поля. Виходячи з того, що системи потокової обробки мають працювати з великими обсягами даних, множинними джерелами, а також з впорядкованими і затриманими подіями, архітектура модуля повинна підтримувати безперервне приймання та уніфікацію вхідного потоку.

Другим елементом є модуль приймання потоку. Його призначення полягає у прийманні повідомлень від зовнішніх джерел, перевірці їх цілісності, початковому розборі структури та передаванні даних в буфер. На цьому етапі виконуються найпростіші операції, яким не треба організації масового паралелізму, тобто перевірку формату, позначення пошкоджених записів, відокремлення корисного навантаження від службової інформації. Для реального часу цей модуль має працювати в неблокуючому режимі, щоб не створювати затримки на вході.

Після приймання дані надходять в буфер пам’яті. Буфер комунікаційною ланкою між асинхронним надходженням сенсорних повідомлень і пакетною передачею підготовлених масивів до GPU. Буферизація потрібна для вирівнювання інтенсивності потоку, формування блоків даних однакового формату та уникнення простоїв графічного процесора. На рівні архітектури буфер поділяється на вхідну область накопичення, область підготовки до копіювання та область вихідних результатів. Така організація спрощує конвеєрну роботу та дає змогу поєднати надходження нових даних з обробкою попередніх пакетів.

Основним центром керування модуля є CPU-частина. На центральному процесорі зосереджена логіка диспетчеризації, керування чергами, контроль стану буфера, запуск ядер CUDA, а також ті операції, які мають нерегулярний характер і не дають суттєвого виграшу при перенесенні на GPU. Це потрібно тому, що CPU залишається основним елементом для багатопотокового керування, роботи з кеш-пам’яттю та розподілу задач між потоками. Для виконання великої кількості однотипних обчислень над масивами даних використано GPU.

Основним обчислювальним блоком архітектури є GPU-ядра CUDA. Тут виконуються масово-паралельні операції над сформованим пакетом сенсорних даних: нормалізація, фільтрація, обчислення похідних характеристик, агрегація, пошук порогових перевищень або інші однотипні дії. Для підвищення ефективності архітектури потрібно також врахувати перекриття передавання даних і виконання ядер за допомогою потоків CUDA та асинхронного копіювання.

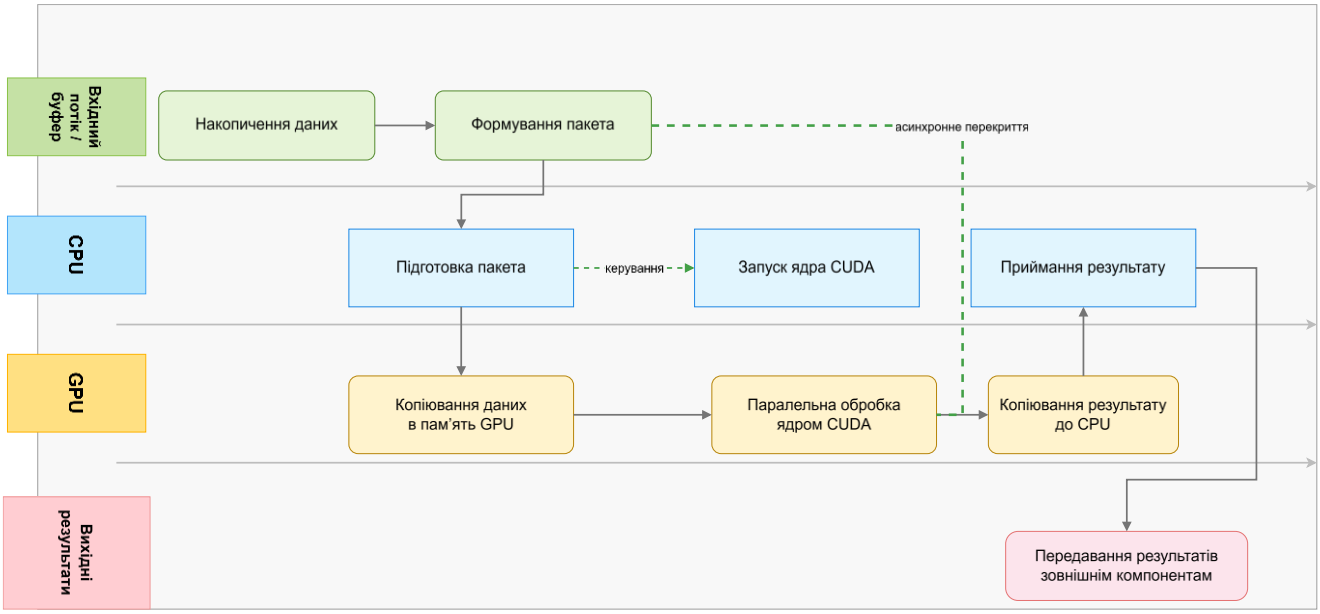


Рисунок 2.2 – Конвеєрна схема обробки потокових сенсорних даних із використанням CPU та GPU

Після завершення обробки активується модуль формування результатів, який приймає вихідні масиви з GPU, виконує постобробку на CPU, перетворює дані у формат, придатний для подальшого використання, та передає їх зовнішнім компонентам. До зовнішніх компонентів відносяться база даних, модуль візуалізації, система моніторингу, API-сервіс, аналітичний модуль та файлова підсистема.

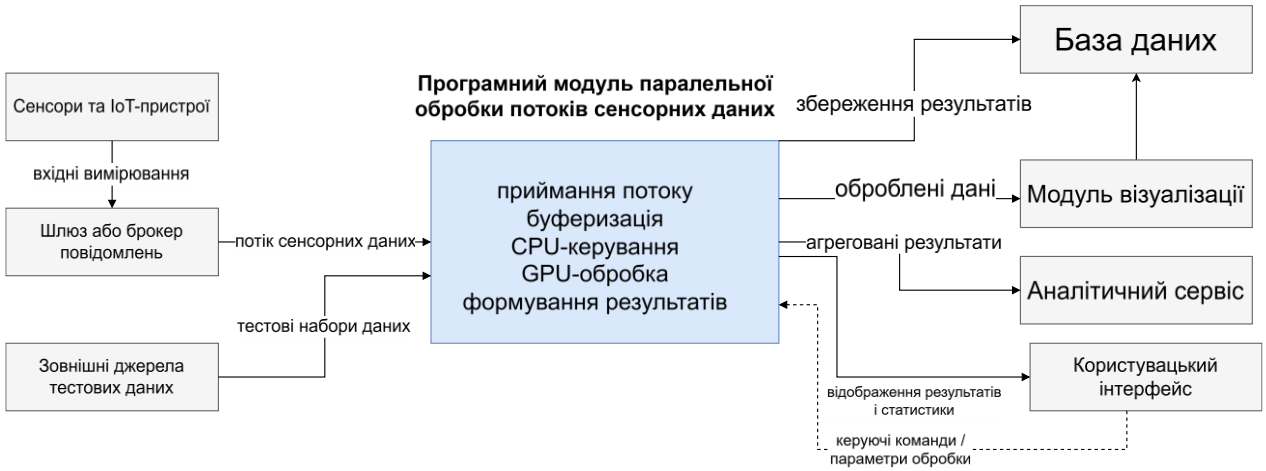


Рисунок 2.3 – Схема інформаційних зв’язків програмного модуля із зовнішніми компонентами системи

Останнім елементом архітектури є інтерфейс взаємодії з іншими компонентами, який має забезпечувати стандартизований обмін даними і відокремлювати внутрішню логіку паралельної обробки від зовнішнього прикладного середовища.

Запропонована архітектура програмного модуля складається з послідовно пов’язаних функціональних частин до яких входять джерела даних, модуль приймання потоку, буфер пам’яті, CPU-частина керування, GPU-ядра CUDA, модуль формування результатів та інтерфейс взаємодії з зовнішніми компонентами. Така структура підходить для потокових систем реального часу, підтримує безперервне надходження даних, розділяє обчислювальні та управляючі функції між CPU та GPU.

2.2 Алгоритми функціонування компонент модуля

З врахуванням призначення розроблюваного програмного модуля його функціонування можна розглядати, як послідовність п’яти взаємопов’язаних алгоритмів, а саме приймання сенсорних потоків, буферизації та підготовки даних, передачі даних з оперативної пам’яті на GPU, паралельної обробки у CUDA, а також повернення результатів і їх подальшого використання.

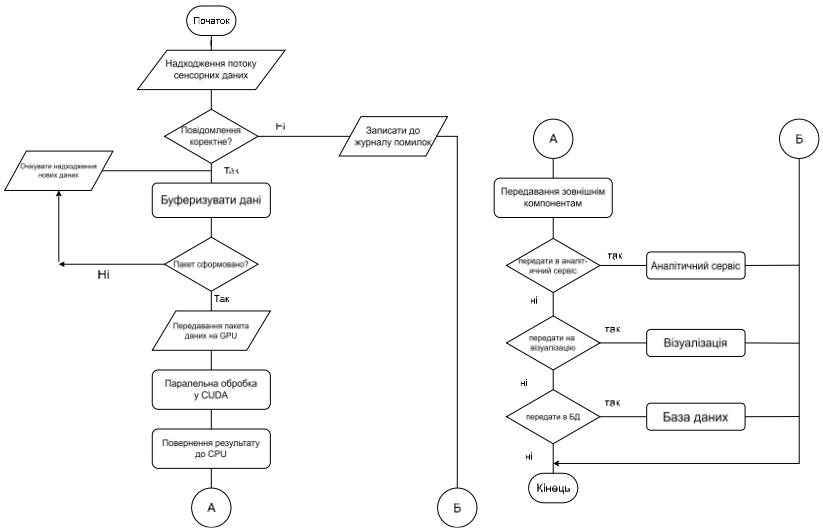


Рисунок 2.4 – Узагальнений алгоритм функціонування програмного модуля паралельної обробки сенсорних потоків

Такий поділ потрібен для потокових систем, в яких вхідний потік спочатку надходить у середовище зберігання або чергу, далі аналізується, передається на обчислювальний рівень і після цього повертається у вигляді вихідних повідомлень або збережених результатів. Такий підхід є класичним рішенням який показано в роботах [37], присвячених потокам даних у туманному та edge-середовищі, де окремо виділяються формування вхідних даних, передача у сховище потоку, обробка, аналіз та генерація вихідних повідомлень.

Першим є алгоритм приймання сенсорних потоків. Його призначення полягає в безперервному отриманні повідомлень від сенсорів, шлюзів або емуляторів, первинній перевірці їх структури та виділенні корисних полів. На даному етапі модуль повинен визначити, чи є повідомлення коректним, чи містить усі обов'язкові атрибути, чи не перевищує допустимий розмір і чи може бути прийняте до подальшої обробки. Якщо повідомлення пошкоджене або неповне, воно має бути позначене як помилкове й або відкинуте, або передане до окремого журналу діагностики. Якщо повідомлення коректне, воно надходить до буфера. Такий алгоритм повинен працювати неблокуючому режимі, щоб інтенсивність надходження даних не призводила до втрати нових повідомлень.

Другим є алгоритм буферизації та підготовки даних. Після приймання, повідомлення накопичуються в буфері до формування пакета. Це потрібно для того, щоб уникнути надмірних накладних витрат на часті передачі невеликих обсягів інформації. В межах цього алгоритму виконується накопичення записів, вирівнювання структури полів, відокремлення службових даних, перетворення типів, а також, за потреби, фільтрація аномальних або явно некоректних значень. Фільтрація аномальних значень і агрегація дозволяють зменшити спотворення результатів і оптимізувати використання обчислювальних ресурсів. Після досягнення встановленого розміру пакета або після закінчення контрольного інтервалу часу сформований, блок передається до CPU-частини керування як готовий до пересилання на GPU.

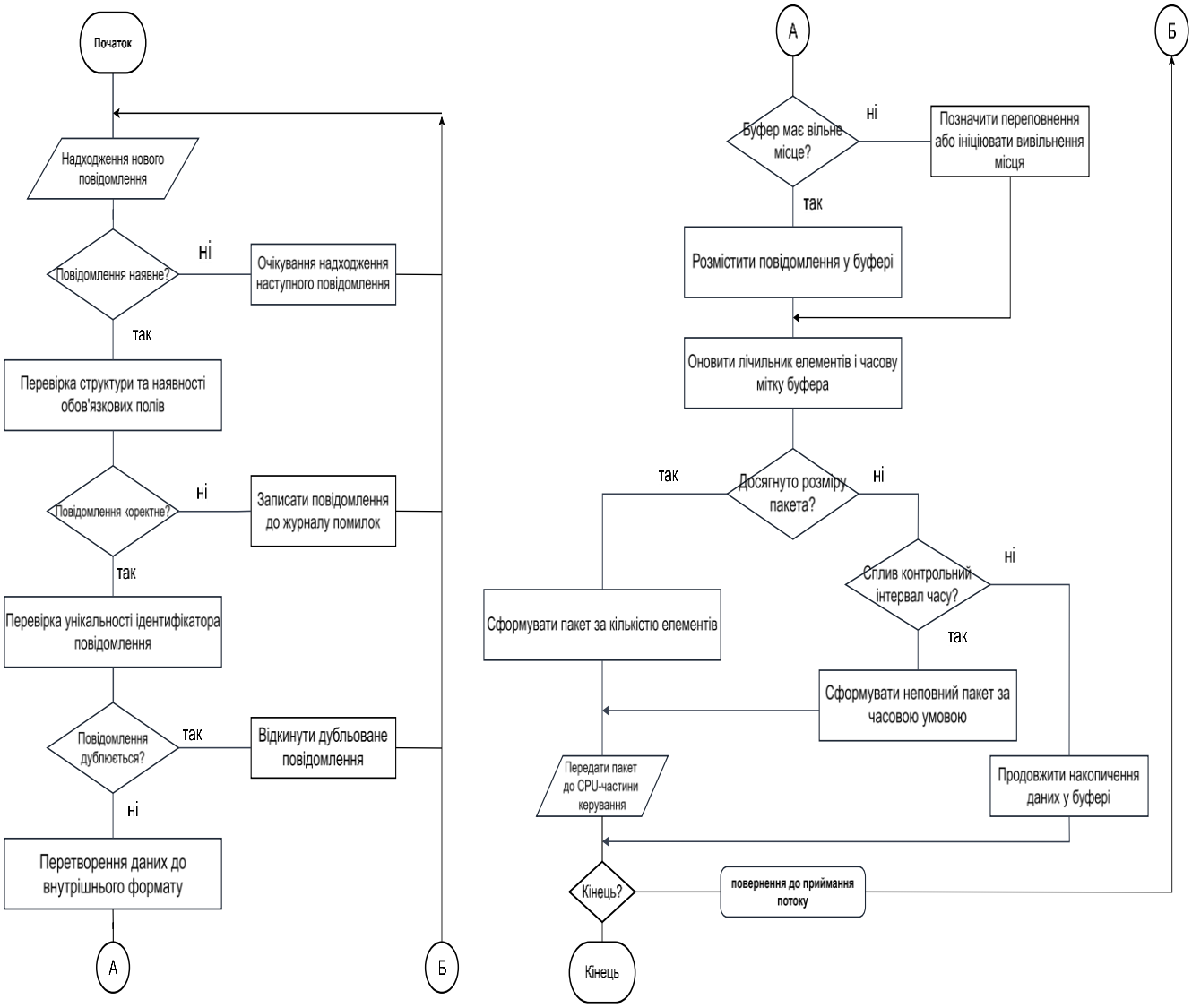


Рисунок 2.5 – Алгоритм приймання, буферизації та підготовки сенсорних даних до обробки

Третім є алгоритм передачі даних з оперативної пам’яті на GPU. Цей етап є одним із критичних для продуктивності, томущо в гетерогенних системах передавання між host і device часто стає вузьким місцем. В роботі [38] з програмування CPU/GPU-систем зазначається, що проблеми передавання даних між центральним процесором і прискорювачем можуть обмежувати перевагу паралелізації.

Тому алгоритм повинен передбачати мінімізацію кількості копіювань, передачу вже підготовлених і компактно розміщених пакетів, а також можливе асинхронне копіювання. На цьому етапі CPU визначає адресу буфера, ініціює передачу в глобальну пам’ять GPU і готує параметри запуску ядра CUDA.

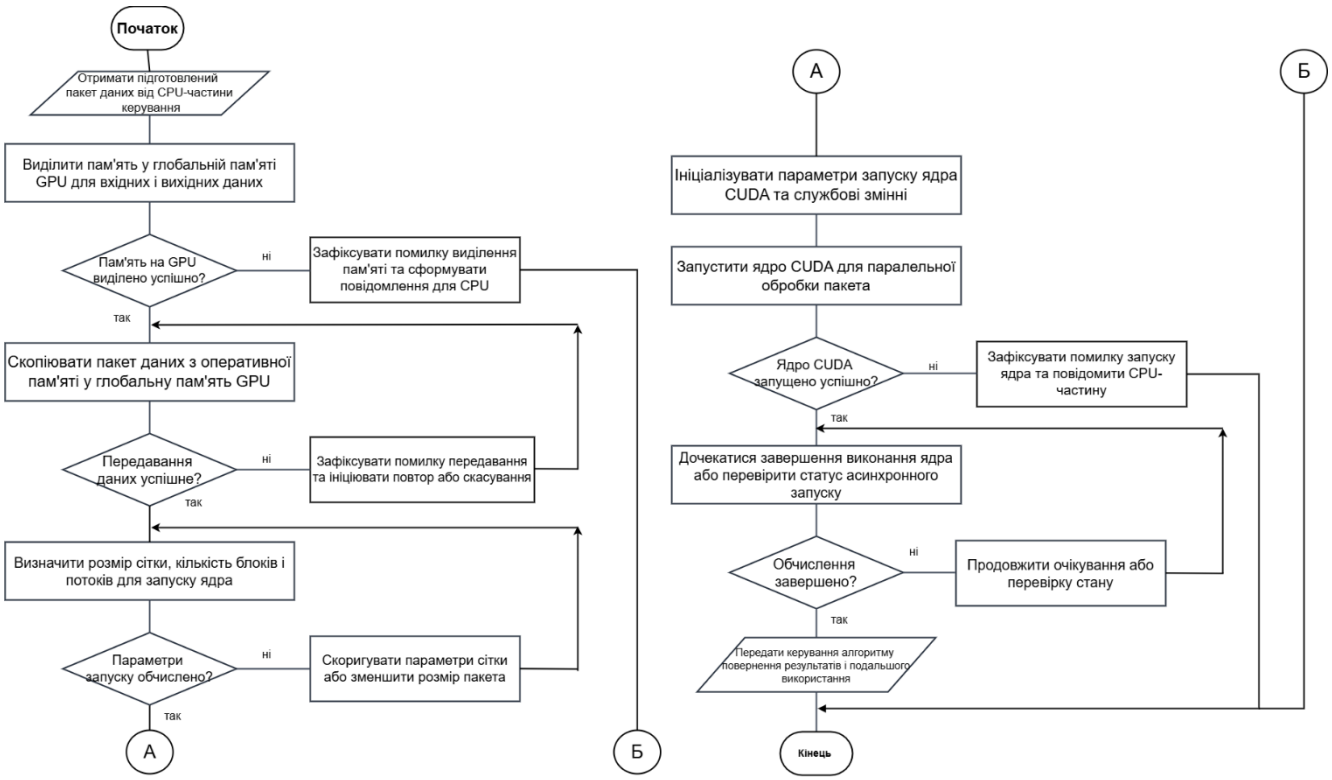


Рисунок 2.6 – Алгоритм передачі пакета даних на GPU та запуску ядра CUDA

Якщо передавання завершується успішно, запускається обчислення якщо є помилка пакет або повторно передається, або відмічається, як проблемний.

Четвертим є алгоритм паралельної обробки у CUDA який забезпечує розв’язання ситуації, що великий пакет однотипних сенсорних записів розподіляється між великою кількістю потоків GPU. Це дозволяє виконувати такі операції, як нормалізація, фільтрація, порогова перевірка, агрегація або обчислення похідних ознак над багатьма записами одночасно.

На рівні алгоритму потрібно передбачити розбиття даних на блоки та сітку, завантаження даних у доступну пам’ять GPU, виконання обраних операцій і запис результату в область вихідного буфера.

Якщо частина обчислень потребує нерегулярного керування, її залишають на CPU, а на GPU передаються тільки ті фрагменти, для яких очікується реальний виграш від паралелізму.

П’ятим є алгоритм забезпечує повернення результатів і їх подальше використання. Після завершення ядра CUDA модуль повинен передати оброблені дані назад до оперативної пам’яті, де CPU виконує, за потреби, постобробку, до

якої входить упорядкування полів, об’єднання частин результату, формування підсумкових структур і підготовка до передавання зовнішнім компонентам. Далі результат може бути спрямований до бази даних, модуля візуалізації, аналітичного сервісу або користувацького інтерфейсу.

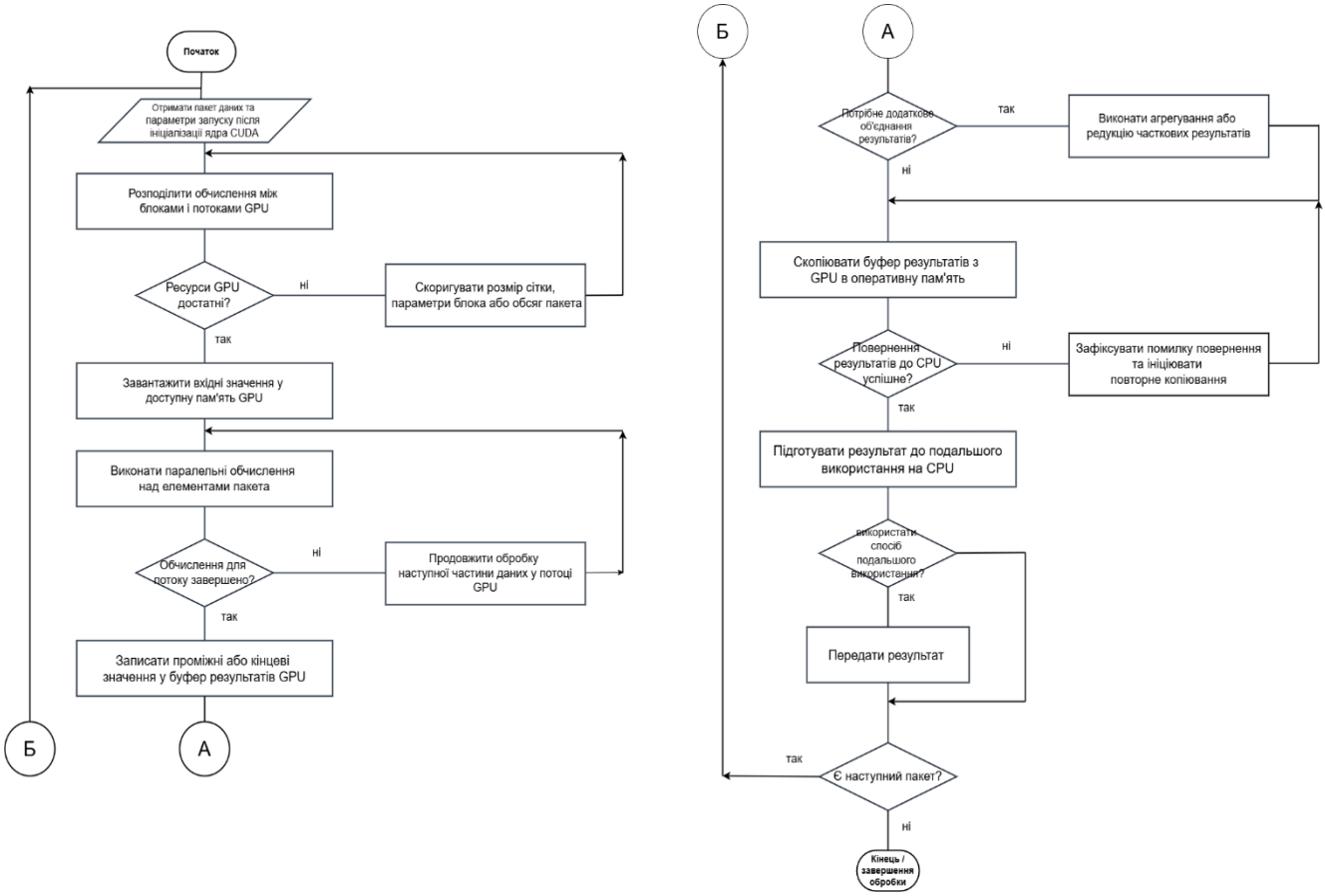


Рисунок 2.7 – Алгоритм паралельної обробки та формування вихідних результатів

Генерація вихідних повідомлень після завершення обробки є завершальним етапом алгоритму, який замикає контур обробки даних.

2.3 Модель взаємодії CPU та GPU при обробці сенсорних потоків

В програмному модулі паралельної обробки сенсорних потоків центральний процесор і графічний прискорювач виконують різні, але взаємопов’язані функції. CPU розглядається, як управляючий рівень системи, який відповідає за приймання

потокових даних, формування пакетів, контроль стану буфера, підготовку параметрів запуску та організацію взаємодії з іншими компонентами.

Графічний процесор в такій моделі виступає обчислювальним рівнем, на який передаються великі масиви однотипних даних для масово-паралельної обробки. Такий підхід дає змогу уникнути дублювання CPU і GPU та добитись розподілу функції відповідно до своїх основних функцій.

Модель взаємодії CPU та GPU при обробці сенсорних потоків складається з декількох етапів.

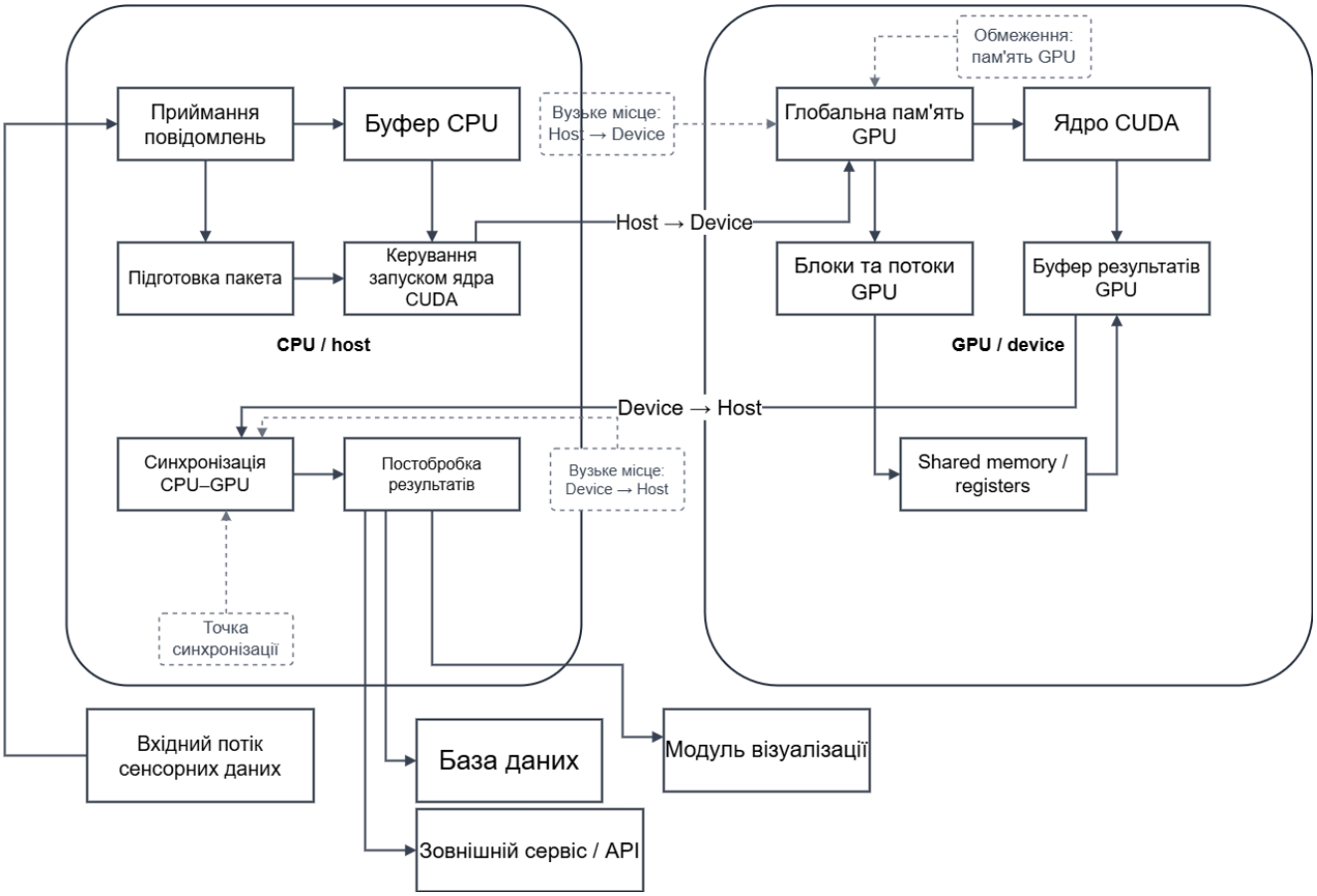


Рисунок 2.8 – Модель взаємодії CPU та GPU при обробці сенсорних потоків

На першому етапі CPU приймає потік повідомлень, виконує їх перевірку, буферизацію та підготовку до передавання. На другому етапі формується пакет даних, який копіюється з оперативної пам’яті host-системи до пам’яті пристрою. На третьому етапі CPU ініціалізує параметри запуску ядра CUDA, виділяється кількість блоків, кількість потоків у блоці та службові параметри обчислення. На

четвертому етапі GPU виконує паралельну обробку пакета. На п’ятому етапі результати копіюються назад у пам’ять CPU, після чого передаються до бази даних, модуля візуалізації або іншого сервісу [39].

$$T_{total} = T_{prep}^{CPU} + T_{H \to D} + T_{kernel} + T_{sync} + T_{D \to H} + T_{post}^{CPU} \quad (2.1)$$

де

– T_{total} — загальний час обробки одного пакета даних;

– T_{prep}^{CPU} — час підготовки пакета на центральному процесорі;

– $T_{H \to D}$ — час копіювання даних з оперативної пам’яті комп’ютера до пам’яті GPU;

– T_{kernel} — час виконання ядра CUDA на графічному процесорі;

– T_{sync} — час синхронізації між CPU та GPU;

– $T_{D \to H}$ — час повернення результатів з пам’яті GPU до оперативної пам’яті комп’ютера;

– T_{post}^{CPU} — час постобробки результатів на центральному процесорі.

Ключовим елементом такої моделі є передавання даних між host і device. Цей етап часто стає одним із головних обмежень продуктивності, тому що передавання даних між host і device в гетерогенних системах продовжує залишатися вузьким місцем, яке обмежує продуктивність. Крім того переміщення даних між CPU та GPU часто займає велику частину часу обчислень, а зменшення цього впливу є використання CUDA streams та cudaMemcpyAsync, що дає змогу перекривати обчислення і передавання даних.

Наступним є внутрішній рівень обчислень на GPU. Потоки групуються в блоки, блоки формують сітку, а самі обчислення виконуються на потокових мультипроцесорах. Така модель означає, що CPU передає на GPU пакет, який далі має бути правильно розкладений між потоками, блоками та рівнями пам’яті. Для сенсорних потоків це важливо, тому що їхня обробка зводиться до виконання

однакових операцій над багатьма записами, тому така структура задачі найкраще відповідає природі CUDA-обчислень.

В моделі також є точки синхронізації. Після запуску ядра CPU має або дочекатися завершення обчислення, або контролювати його стан в разі асинхронного виконання. Після завершення роботи GPU виконується повернення результатів до оперативної пам’яті, а далі — їх підготовка для подальшого використання. Підвищення ефективності досягається ще за рахунок того організовується правильне поєднання етапів копіювання, обчислення, синхронізації та повернення результатів.

Вузькі місця в цій моделі можна звести до кількох груп куди входять витрати часу на копіювання даних між CPU і GPU, обмеження обсягу пам’яті GPU, через які формуються пакети оптимального розміру та схеми часткової обробки. Крім того є затримки, пов’язані з глобальною пам’яттю, неефективним доступом до неї та конфліктами під час роботи з shared memory. Крім того є необхідність балансування між розміром пакету, кількістю потоків і кількістю блоків, тому що малий розмір пакету не дає достатнього паралелізму, а великий може створювати навантаження на пам’ять і збільшувати час передавання [40].

$$P = \frac{N}{T_{total}} \tag{2.2}$$

де P— пропускна здатність;

N— кількість записів у пакеті;

T_{total} — загальний час обробки пакету.

Тому модель взаємодії CPU та GPU при обробці сенсорних можна розглядати як узгоджену багатоступеневу схему, в якій CPU виконує керуючі та підготовчі функції, а GPU — основну масово-паралельну обробку даних. Ефективність такої моделі визначається швидкодією ядра CUDA та організацією всієї траєкторії проходження пакету від буферизації та копіювання до синхронізації, повернення результатів і передавання їх зовнішнім компонентам [41].

$$S = \frac{T_{CPU}}{T_{CPU+GPU}} \quad (2.3)$$

де S — коефіцієнт прискорення;

T_{CPU} — час виконання задачі лише на CPU;

$T_{CPU+GPU}$ — час виконання задачі при використанні CPU та GPU.

Розглянута модель взаємодії CPU та GPU показує архітектуру програмного модуля, подану в підрозділі 2.1, та розкриває логіку алгоритмів, описаних у підрозділі 2.2. В межах цієї моделі визначено, які потрібно виконувати на центральному процесорі, а які — передавати на графічний прискорювач для масово-паралельної обробки. Це важливо тому, що ефективність програмного модуля залежить від швидкодії ядра CUDA та від узгодженості етапів приймання потоку, буферизації, підготовки пакета, передавання даних між CPU і GPU, виконання обчислень, синхронізації та повернення результатів. Побудована модель є основою для реалізації програмного модуля паралельної обробки сенсорних потоків і для оцінювання його продуктивності.

3 ПРОГРАМНО-ТЕХНОЛОГІЧНЕ ЗАБЕЗПЕЧЕННЯ МОДУЛЯ

3.1 Вибір програмних і технічних засобів реалізації

Для реалізації програмного модуля паралельної обробки сенсорних потоків було вибрано наступні програмні та технічні засоби, які, забезпечують підтримку технології CUDA та дозволяють реалізувати повний цикл роботи модуля: приймання вхідного потоку, буферизацію, підготовку пакетів, запуск обчислень на GPU, повернення результатів і їх подальший аналіз. Вибір засобів реалізації повністю відповідає архітектурі модуля, який було представлено в другому розділі, та з моделлю взаємодії CPU і GPU, тому що ці компоненти визначають логіку програмної реалізації.

Основною мовою програмування та реалізації вибрано C++ у поєднанні з CUDA C/C++. Такий вибір обумовлений тим, що C++ надає розробнику прямий доступ до пам’яті, структур даних, механізмів синхронізації та засобів керування продуктивністю.

Для задачі потокової обробки це є основним, тому що модуль повинен працювати з чергами повідомлень, буферами, пакетами даних і взаємодією між CPU та GPU. Використання CUDA C/C++ дозволяє реалізувати ядра графічного процесора безпосередньо в тій самій програмній моделі, в якій побудована host-частина модуля. В результаті цього спрощується зв’язок між CPU-частиною, що відповідає за приймання потоку та керування, і GPU-частиною, що виконує масово-паралельні обчислення.

Основним інструментом для реалізації GPU-частини вибрано NVIDIA CUDA Toolkit. Для наявної системи з NVIDIA GeForce 940MX основний фокус був на гілці CUDA 12.x, оскільки зазначено, що починаючи з CUDA 13.0 для архітектур Maxwell, Pascal і Volta вилучено offline compilation та library support, тоді як використання серії 12.x для побудови застосунків під ці архітектури залишається підтримуваним. Це робить CUDA 12.x найбільш придатним вибором для практичної реалізації модуля на реальній доступній апаратній платформі.

В якості середовища розробки використано Microsoft Visual Studio 2022 Community. Такий вибір зумовлений тим, що Visual Studio входить до переліку підтримуваних компіляторів в документації NVIDIA для CUDA на Windows.

Для організації потокової логіки всередині модуля (рисунок 3.1) використано стандартні бібліотеки C++, такі як thread, mutex, condition_variable, queue, vector, chrono, fstream. Їх функціональності вистачає для реалізації приймання потоку, буферизації, формування пакетів, внутрішніх черг і синхронізації між окремими етапами обробки. Такий вибір пояснюється тим, що основною розробкою є локальний модуль, а не розподілена система потокової аналітики корпоративного рівня. Тому використання важких зовнішніх платформ ускладнило саму реалізацію. Для GPU-частини застосовано CUDA Runtime API, зокрема засоби роботи з пам’яттю, запуску ядер, синхронізації та асинхронного передавання даних.

Крім того також можна використати Python, як допоміжний інструмент.

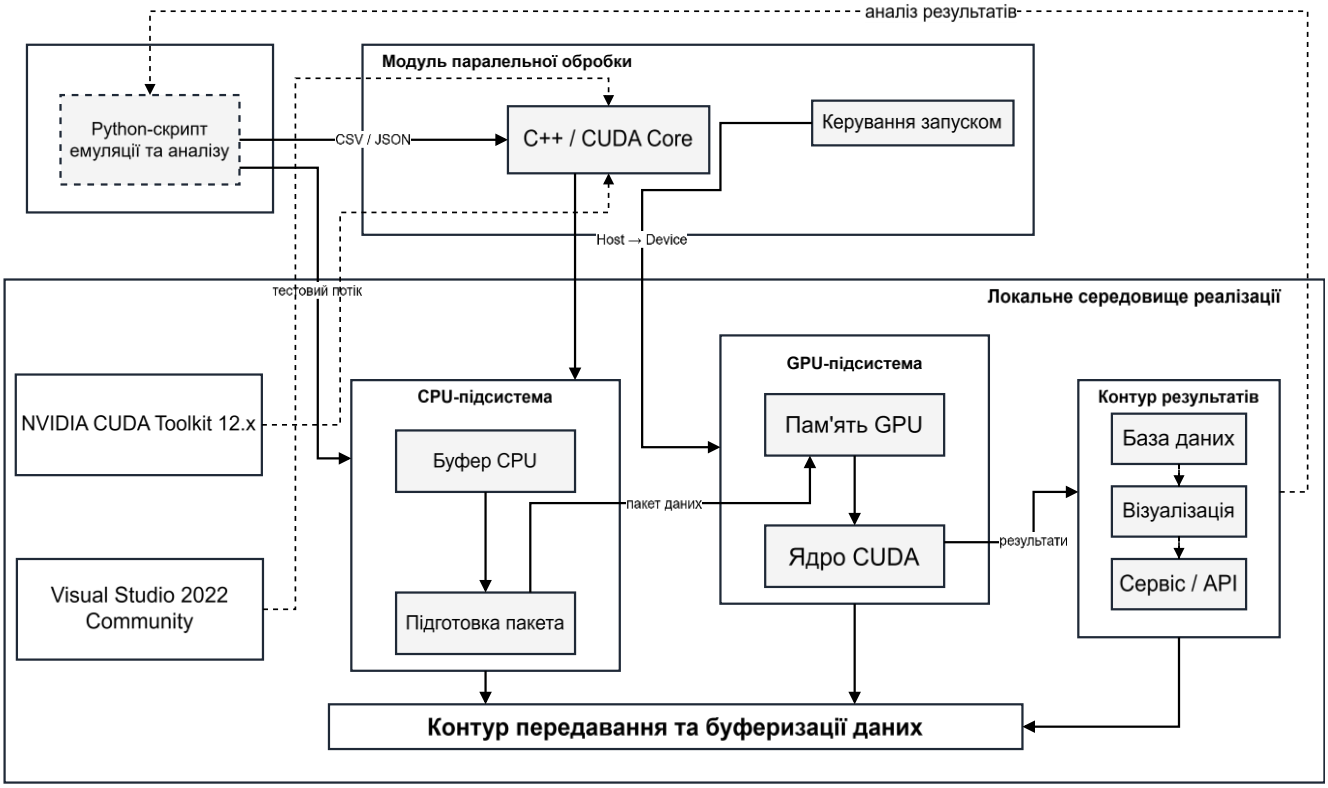


Рисунок 3.1 – Програмний контур реалізації модуля паралельної обробки сенсорних потоків

Такий підхід є найбільш раціональним, тому що основний модуль залишається реалізованим мовою C++ з використанням CUDA, а Python застосовується там, де він справді зручний: для генерації або емуляції вхідних сенсорних потоків, підготовки тестових наборів даних, запуску серій експериментів та аналізу отриманих результатів.

Як тип вхідних даних буде використано 2 типи – це емульований потік сенсорних повідомлень та потік від реальної системи вімірювання на базі модуля esp32.

Такі дані зручно подавати у вигляді структурованих записів CSV- або JSON-подібного формату, де кожен запис містить ідентифікатор сенсора, часову мітку та один або кілька виміряних параметрів. Емуляція потоку може здійснюватися Python-скриптом, який або відтворює наперед підготовлений набір записів з певною частотою, або генерує синтетичні дані в режимі реального часу.

В якості технічної платформи використано Windows-машину з графічним адаптером NVIDIA GeForce 940MX. Цей графічний процесор не належить до сучасних високопродуктивних рішень, він підтримує CUDA. Це дає можливість реалізувати базові CUDA-ядра, дослідити взаємодію CPU та GPU, а також провести порівняння часу виконання CPU- і GPU-варіантів на одній машині.

Такий набір засобів є достатнім для створення прототипу модуля, проведення експериментів і подальшого оцінювання ефективності паралельної обробки сенсорних потоків.

3.2 Програмна реалізація модуля обробки сенсорних даних

Основним компонентом програмної реалізації розробленого модуля є файл sensor_cuda.cu (Додаток В), в якому реалізовано обробку вхідних сенсорних даних у двох режимах: послідовному на центральному процесорі та паралельному на графічному процесорі з використанням технології CUDA. Це дає змогу виконувати цільову обробку даних та безпосередньо порівнювати результати і час виконання для CPU- і GPU-варіантів на одному й тому самому наборі вхідних значень.

На початковому етапі модуль зчитує вхідні дані з файлу input.csv (Додаток В). Для цього використано окрему функцію читання, яка послідовно опрацьовує рядки файлу, пропускає заголовок і перетворює кожний запис у внутрішню структуру SensorRecord, що містить ідентифікатор запису та його числове значення. Після зчитування дані переносяться у вектор вхідних значень, який надалі використовується як джерело для обчислень.

В модулі передбачено два режими обробки. В режимі cpu використовується окрема функція послідовного проходу по масиву вхідних значень. Для кожного елемента виконується нормалізація шляхом ділення на 100 та визначення бінарного прапорця перевищення порогового значення. В режимі gpu аналогічні дії виконує CUDA-ядро processSensorDataGpu, яке запускається для сітки потоків, де кожен потік обробляє один елемент масиву. Перед запуском ядра виконується виділення пам’яті на GPU, копіювання вхідних даних з оперативної пам’яті до пам’яті пристрою, а після завершення обчислень — зворотне копіювання нормалізованих значень і прапорців у пам’ять CPU.

Після основних операцій cudaMalloc, cudaMemcpy, запуску ядра та синхронізації перевіряється статус виконання. Це дозволяє виявити можливі помилки ще на етапі тестування і забезпечує надійну роботу програмного модуля. Для вимірювання продуктивності використано засоби стандартної бібліотеки chrono, за допомогою яких визначається час обробки в мілісекундах.

Після завершення основної обробки модуль формує узагальнену статистику для поточного набору даних. Визначаються мінімальне та максимальне значення, середнє значення вибірки, кількість записів, що перевищили заданий поріг, а також частка таких записів у відсотках. Отримані результати записуються у файл output.csv, де для кожного запису зберігаються ідентифікатор, початкове значення, нормалізоване значення та прапорець перевищення порога. Крім того, модуль додає узагальнений рядок у файл benchmark.csv, який використовується як журнал експериментів і містить режим обробки, кількість записів, поріг, статистичні характеристики та час виконання.

Файл sensor_cuda.cu є ядром розробленої системи, забезпечує, функціональну обробку сенсорних значень, так і експериментальне порівняння послідовного та паралельного способів виконання.

Для забезпечення взаємодії користувача з розробленим модулем було створено простий графічний інтерфейс у файлі gui.py. Його реалізовано мовою Python із використанням стандартної бібліотеки Tkinter.

Інтерфейс (рисунок 3.2) виконує роль проміжної ланки між користувачем і основним модулем обробки sensor_cuda.exe. У вікні програми передбачено поле для введення кількості записів, поле для задання порогового значення та перемикач режиму обробки, який дозволяє вибирати між CPU- та GPU-виконанням. Крім цього, реалізовано дві основні кнопки: одна відповідає за генерацію вхідного набору даних, а друга — за запуск обробки. Нижня частина вікна містить текстове поле, в якому відображається службова інформація, куди записуються результат виконання програми та перші рядки вихідного файлу.

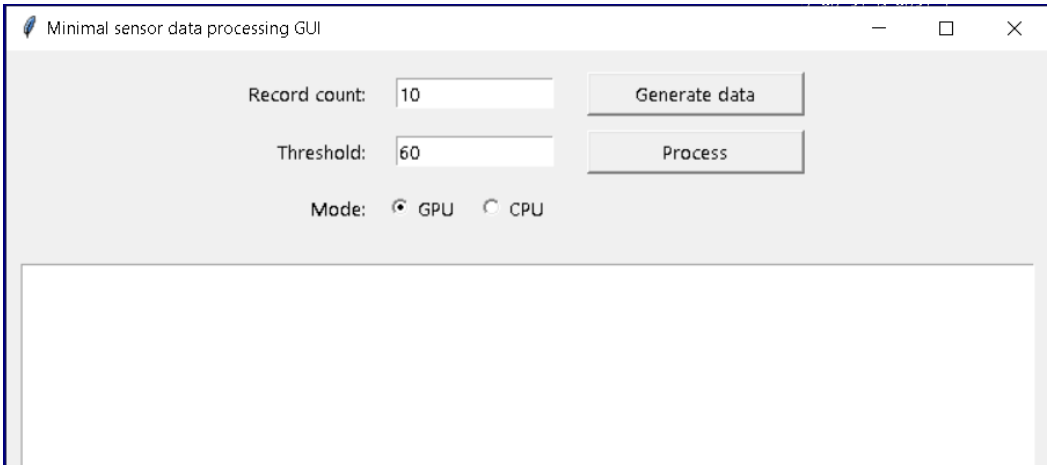


Рисунок 3.2 – Головне вікно графічного інтерфейсу програмного модуля

Під час натискання кнопки генерації даних інтерфейс формує файл input.csv. Для цього використовується вбудований генератор випадкових чисел, який створює задану користувачем кількість значень у діапазоні від 0 до 100. Кожному запису надається власний ідентифікатор, після чого дані записуються у CSV-файл із заголовком.

Під час натискання кнопки обробки програма спочатку перевіряє наявність виконуваного файлу sensor_cuda.exe та файлу input.csv. Якщо один із них відсутній, користувач отримує відповідне повідомлення. Після цього інтерфейс формує команду запуску, у якій передає порогове значення та вибраний режим роботи. Запуск основного модуля здійснюється за допомогою subprocess.run, тобто графічний інтерфейс не виконує обчислення самостійно, а передає ці дії зовнішньому виконуваному файлу. Така організація є простою та зручною, оскільки дозволяє чітко розділити функції: Python відповідає за керування, а CUDA-модуль — за обчислення.

Після завершення роботи sensor_cuda.exe графічний інтерфейс приймає текстовий вивід програми та відображає його у вікні. Це дає можливість одразу бачити режим обробки, кількість записів, поріг, статистичні характеристики, кількість перевищень порога та час виконання. Додатково інтерфейс відкриває файл output.csv і виводить кілька перших рядків результату, що дозволяє швидко перевірити правильність обробки без необхідності окремо відкривати файл у сторонньому редакторі.

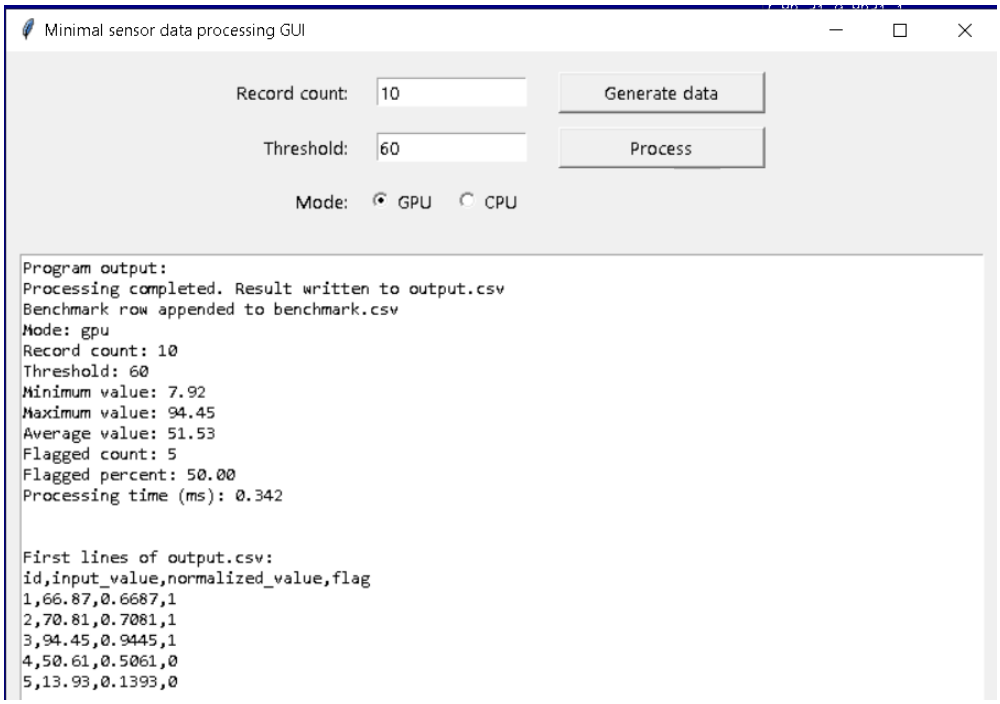


Рисунок 3.3 – Вікно програми після виконання обробки сенсорних даних

Файл gui.py реалізує мінімальний графічний інтерфейс користувача, достатній для генерації вхідних даних, запуску обробки в різних режимах і перегляду результатів..

3.3 Дослідження ефективності програмного модуля

Після реалізації програмного модуля паралельної обробки сенсорних потоків і створення мінімального графічного інтерфейсу було виконано його експериментальну перевірку. Метою тестування було підтвердження коректності роботи розробленої системи, перевірка збіжності результатів для послідовного та паралельного режимів, а також оцінювання часу обробки вхідних даних для різних обсягів потоку. Так як в межах роботи не використовувався реальний набір фізичних сенсорів, бо їх поведінку дуже легко змоделювати, вхідні значення формувалися програмно у вигляді штучно згенерованих записів, що зберігалися у файлі input.csv. Це дозволило легко змінювати кількість елементів у пакеті, повторювати експерименти та порівнювати результати в однакових умовах.

Під час тестування для кожного набору даних виконувалася одна й та сама послідовність дій. Спочатку за допомогою графічного інтерфейсу формувався вхідний файл із заданою кількістю значень. Після цього запускалася обробка в режимі cpu або gpu, а результати зберігалися у файлах output.csv і benchmark.csv. В процесі виконання програма визначала нормалізовані значення, формувала прапорець перевищення порога, обчислювала мінімальне, максимальне та середнє значення вибірки, а також кількість і відсоток записів, що перевищили встановлений поріг. Для всіх проведених перевірок використовувалося порогове значення 60. Такий режим тестування дозволив одночасно оцінити як функціональну правильність роботи модуля, так і його часові характеристики.

Перший тест було виконано для невеликого пакета з 10 записів. Для такого обсягу даних режим CPU показав час виконання 0,000 мс, тоді як в режимі GPU час становив 0,420 мс. Цей результат є очікуваним, оскільки для дуже малих пакетів накладні витрати є більшими, ніж сама тривалість обчислень. Отже, для малих

вхідних наборів використання GPU не дає виграшу в продуктивності, хоча обидва режими формують однаковий результат.

Другий тест було проведено для пакета обсягом 100000 записів (рисунок 3.4). В цьому випадку для одного й того самого набору даних у режимі CPU було отримано час 1,472 мс, а в режимі GPU — 1,236 мс. Інші характеристики збіглися повністю: мінімальне значення становило 0,00, максимальне — 100,00, середнє — 50,12, кількість значень, що перевищили поріг, — 40151, а їх частка — 40,15 %. Це означає, що при зростанні обсягу даних паралельна обробка вже починає демонструвати перевагу над послідовною.

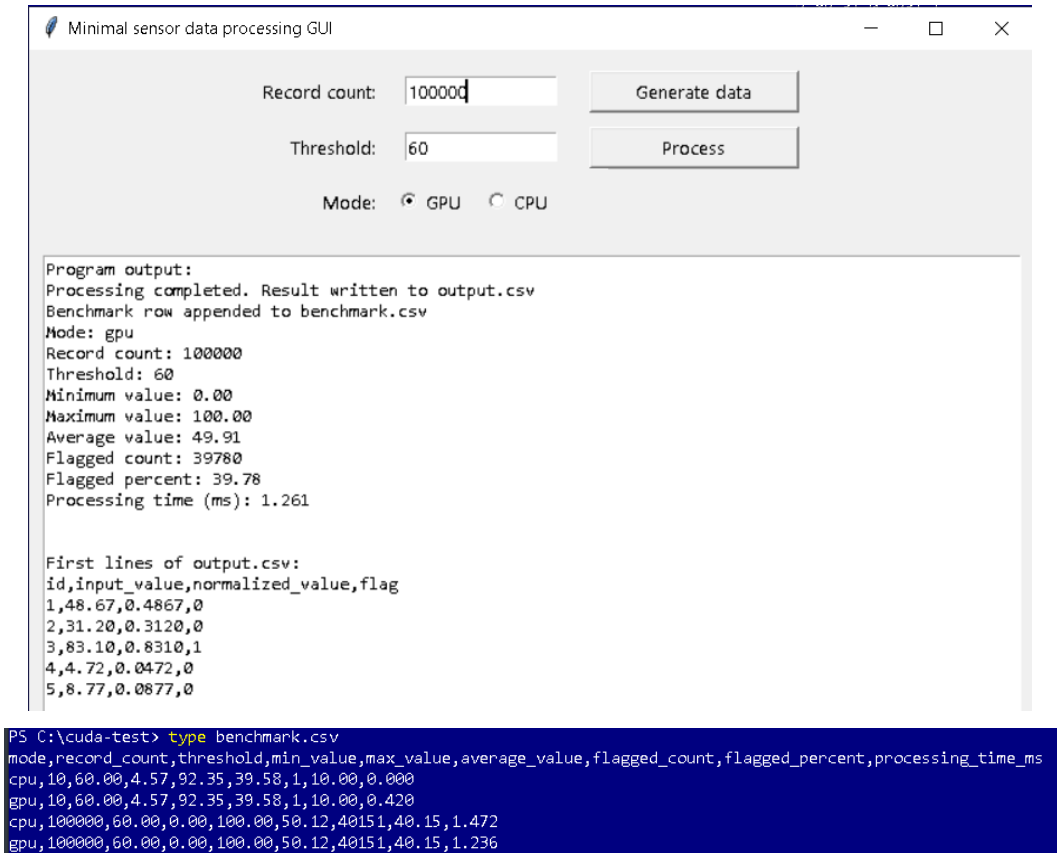


Рисунок 3.4 – Результат експерименту для пакета з 100000 записів

Третій тест було виконано для пакета розміром 1000000 записів. Для цього набору в режимі CPU час обробки становив 14,880 мс, тоді як у режимі GPU — 9,605 мс. При цьому знову було отримано однакові статистичні показники: мінімальне значення 0,00, максимальне 100,00, середнє 50,01, кількість перевищень порога 399822, а відсоток таких значень 39,98 %. Отримані результати свідчать, що

зі збільшенням кількості даних ефективність GPU-обробки стає більш помітною. У цьому тесті GPU виконував обробку приблизно у 1,55 рази швидше, ніж CPU (рисунок 3.5).

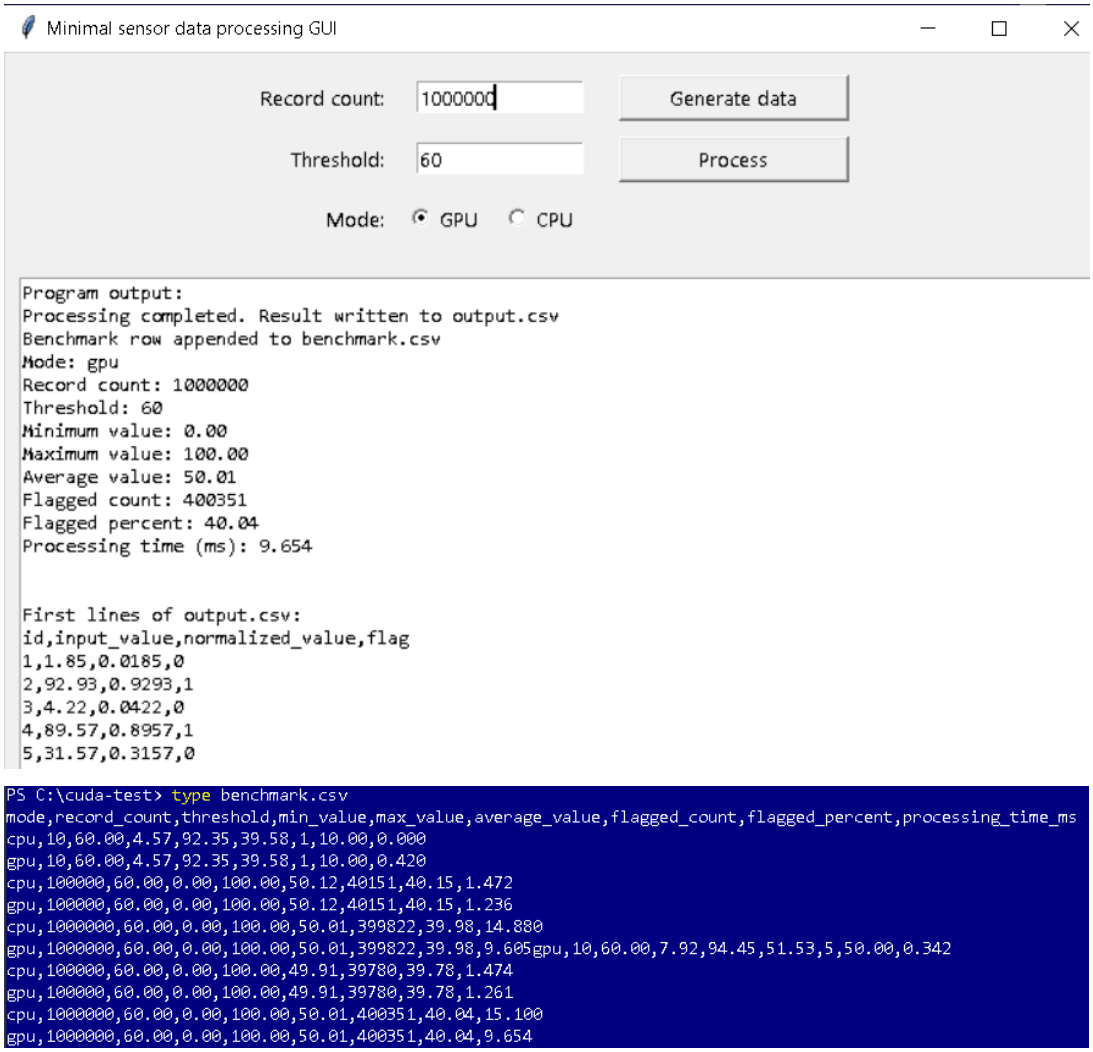


Рисунок 3.5 – Результат експерименту для пакета з 1000000 записів

Проведене тестування дозволяє зробити кілька висновків: а)розроблений програмний модуль працює коректно, так як для однакових вхідних даних режими CPU та GPU формують однакові значення на виході, б)для малих пакетів використання графічного процесора не є доцільним через суттєві накладні витрати. в) із збільшенням кількості записів перевага GPU-обробки зростає. Отже, розроблений модуль може використовуватись як експериментальний прототип для опрацювання великих масивів сенсорних даних.

ВИСНОВКИ

При виконанні кваліфікаційної роботи на тему «Програмний модуль паралельної обробки сенсорних потоків IoT на базі технології CUDA» було досягнуто наступних результатів:

1. Проведено аналіз особливостей сенсорних потоків даних в IoT-системах.

2. Проаналізовано сучасні методи і засоби паралельної обробки даних, зокрема CPU-орієнтовані та GPU-орієнтовані підходи.

3. Проведено аналіз існуючих програмних рішень і платформ для потокової обробки IoT-даних.

4. Сформульовано постановку задачі, яка полягає у створенні програмного модуля паралельної обробки сенсорних потоків IoT.

5. Розроблено архітектуру програмного модуля паралельної обробки сенсорних потоків. Визначено основні складові модуля, до яких входять джерела даних, модуль приймання потоку, буфер пам’яті, CPU-частина керування, GPU-ядра CUDA, модуль формування результатів та інтерфейс взаємодії із зовнішніми компонентами системи.

6. Розроблено алгоритми функціонування основних компонентів модуля.

7. Розроблено інформаційне забезпечення програмного модуля та модель взаємодії CPU і GPU при обробці сенсорних потоків.

8. Проведено вибір програмних і апаратних засобів реалізації. Для створення модуля обрано мову C++ у поєднанні з CUDA C/C++, NVIDIA CUDA Toolkit, середовище Microsoft Visual Studio 2022, стандартні бібліотеки C++ для організації потокової логіки та Python як допоміжний засіб для генерації даних.

9. Реалізовано ключові модулі програмної системи. Створено основний модуль sensor_cuda.cu, у якому реалізовано режими обробки на CPU та GPU, нормалізацію даних, порогову перевірку, формування статистики, запис результатів у output.csv та журналу експериментів у benchmark.csv.

10. Проведено тестування працездатності програмного модуля та оцінено його ефективність на наборах із 10, 100000 і 1000000 записів, час CPU-обробки

становив 14,880 мс, а GPU-обробки — 9,605 мс, тобто GPU виконував обробку приблизно у 1,55 рази швидше.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Krishnamurthi R., Kumar A., Gopinathan D., Nayyar A., Qureshi B. An Overview of IoT Sensor Data Processing, Fusion, and Analysis Techniques. Sensors. 2020. Vol. 20, No. 21. Art. 6076

2. Abdulhussain S. H., Mahmmod B. M., Alwhelat A., Shehada D., Shihab Z. I., Mohammed H. J. et al. A Comprehensive Review of Sensor Technologies in IoT: Technical Aspects, Challenges, and Future Directions. Computers. 2025. Vol. 14, No. 8. Art. 342.

3. RIoTBench: A Real-time IoT Benchmark for Distributed Stream Processing Platforms, Anshu Shukla, Shilpa Chaturvedi and Yogesh Simmhan, Concurrency and Computation: Practice and Experience, Volume 29, Issue 21, 2017, doi:10.1002/cpe.4257

4. Shukla A., Chaturvedi S., Simmhan Y. RIoTBench: A Real-time IoT Benchmark for Distributed Stream Processing Platforms. arXiv preprint. 2017. arXiv:1701.08530v1.

5. Krishnamurthi R., Kumar A., Gopinathan D., Nayyar A., Qureshi B. An Overview of IoT Sensor Data Processing, Fusion, and Analysis Techniques. Sensors. 2020. Vol. 20, No. 21. Art. 6076.

6. Abdulhussain S. H., Mahmmod B. M., Alwhelat A., Shehada D., Shihab Z. I., Mohammed H. J. et al. A Comprehensive Review of Sensor Technologies in IoT: Technical Aspects, Challenges, and Future Directions. Computers. 2025. Vol. 14, No. 8. Art. 342.

7. Krishnamurthi R., Kumar A., Gopinathan D., Nayyar A., Qureshi B. An Overview of IoT Sensor Data Processing, Fusion, and Analysis Techniques. Sensors. 2020. Vol. 20, No. 21. Art. 6076.

8. Мальченко Д. С. Метод збору та зберігання інформації сенсорної мережі Інтернету речей: дипломна робота бакалавра. Київ: НТУУ «КПІ імені Ігоря Сікорського», 2025.

9. Shukla A., Chaturvedi S., Simmhan Y. RIoTBench: A Real-time IoT Benchmark for Distributed Stream Processing Platforms. arXiv preprint. 2017. arXiv:1701.08530v1.

10. Hamdan S., Ayyash M., Almajali S. Edge-Computing Architectures for Internet of Things Applications: A Survey. Sensors. 2020. Vol. 20, No. 22. Art. 6441.

11. Резанов Б. М. Моделі та методи оптимізації розподілу завдань у туманному середовищі Інтернету речей: дис. ... д-ра філософії за спец. 123 – Комп'ютерна інженерія. Харків: НТУ «Харківський політехнічний інститут», 2023.

12. Gkonis P., Giannopoulos A., Trakadas P., Masip-Bruin X., D'Andria F. A Survey on IoT-Edge-Cloud Continuum Systems: Status, Challenges, Use Cases, and Open Issues. Future Internet. 2023. Vol. 15. Art. 383.

13. Pacella M., Papa A., Papadia G., Fedeli E. A Scalable Framework for Sensor Data Ingestion and Real-Time Processing in Cloud Manufacturing. Algorithms. 2025. Vol. 18, No. 1. Art. 22.

14. Krishnamurthi R., Kumar A., Gopinathan D., Nayyar A., Qureshi B. An Overview of IoT Sensor Data Processing, Fusion, and Analysis Techniques. Sensors. 2020. Vol. 20, No. 21. Art. 6076.

15. Shukla A., Chaturvedi S., Simmhan Y. RIoTBench: A Real-time IoT Benchmark for Distributed Stream Processing Platforms. arXiv preprint. 2017. arXiv:1701.08530v1.

16. Gkonis P., Giannopoulos A., Trakadas P., Masip-Bruin X., D'Andria F. A Survey on IoT-Edge-Cloud Continuum Systems: Status, Challenges, Use Cases, and Open Issues. Future Internet. 2023. Vol. 15. Art. 383.

17. Pharr M., Mark W. R. ispc: A SPMD Compiler for High-Performance CPU Programming. 2012. https://pharr.org/matt/assets/ispc.pdf

18. ALHafez N., Kurdi A. Parallel Paradigms in Modern HPC: A Comparative Analysis of MPI, OpenMP, and CUDA. arXiv, 2025. https://arxiv.org/pdf/2506.15454

19. ALHafez N., Kurdi A. Parallel Paradigms in Modern HPC: A Comparative Analysis of MPI, OpenMP, and CUDA. arXiv, 2025. https://arxiv.org/pdf/2506.15454

20. Liu G. et al. MIMO Radar Parallel Simulation System Based on CPU/GPU Architecture. Sensors, 2022. https://www.mdpi.com/1424-8220/22/1/396

21. ALHafez N., Kurdi A. Parallel Paradigms in Modern HPC: A Comparative Analysis of MPI, OpenMP, and CUDA. arXiv, 2025. https://arxiv.org/pdf/2506.15454

22. ALHafez N., Kurdi A. Parallel Paradigms in Modern HPC: A Comparative Analysis of MPI, OpenMP, and CUDA. arXiv, 2025. https://arxiv.org/pdf/2506.15454

23. Liu G. et al. MIMO Radar Parallel Simulation System Based on CPU/GPU Architecture. Sensors, 2022. https://www.mdpi.com/1424-8220/22/1/396

24. Pharr M., Mark W. R. ispc: A SPMD Compiler for High-Performance CPU Programming. 2012. https://pharr.org/matt/assets/ispc.pdf

25. Pharr M., Mark W. R. ispc: A SPMD Compiler for High-Performance CPU Programming. 2012. https://pharr.org/matt/assets/ispc.pdf

26. ALHafez N., Kurdi A. Parallel Paradigms in Modern HPC: A Comparative Analysis of MPI, OpenMP, and CUDA. arXiv, 2025. https://arxiv.org/pdf/2506.15454

27. ALHafez N., Kurdi A. Parallel Paradigms in Modern HPC: A Comparative Analysis of MPI, OpenMP, and CUDA. arXiv, 2025. https://arxiv.org/pdf/2506.15454

28. Авраменко А. В. Засоби CUDA для оброблення матричних даних великих розмірів: дипломна робота бакалавра. КПІ ім. Ігоря Сікорського, 2021. https://ela.kpi.ua/items/cfcbe5e1-a7ee-470b-a390-d980ba498792

29. Ding L., Dong Z., He H., Zheng Q. A Hybrid GPU and CPU Parallel Computing Method to Accelerate Millimeter-Wave Imaging. Electronics, 2023. https://www.mdpi.com/2079-9292/12/4/840

30. Liu G. et al. MIMO Radar Parallel Simulation System Based on CPU/GPU Architecture. Sensors, 2022. https://www.mdpi.com/1424-8220/22/1/396

31. Almeida A. et al. A survey on data stream frameworks, analysis and algorithms. Journal of Big Data, 2023. https://journalofbigdata.springeropen.com/articles/10.1186/s40537-023-00760-1

32. Степанюк Є. Ю. Математичне та програмне забезпечення для аналізу потоків текстових даних. https://ela.kpi.ua/bitstream/123456789/31665/1/Stepaniuk_magistr.pdf

33. Liu F. et al. A Survey on Edge Computing Systems and Tools. Proceedings of the IEEE / arXiv, 2019. https://arxiv.org/abs/1911.02794

34. Ang L.-M. et al. GPU-Based Embedded Intelligence Architectures and Platforms for Internet of Things: A Survey. Electronics, 2021. https://www.mdpi.com/2079-9292/10/8/952

35. Zhang S. et al. Hardware-Conscious Stream Processing: A Survey. arXiv, 2020. https://arxiv.org/pdf/2001.05667

36. Yi X. A Study of Performance Programming of CPU, GPU Accelerated Computers and SIMD Architecture. arXiv, 2024. https://arxiv.org/pdf/2409.10661

37. Krishnamurthi R. et al. An Overview of IoT Sensor Data Processing, Fusion, and Analysis Techniques], [Hamdan S., Ayyash M., Almajali S. Edge-Computing Architectures for Internet of Things Applications: A Survey

38. Xinyao Yi, A Study of Performance Programming of CPU, GPU accelerated Computers and SIMD Architecture.

39. Van Werkhoven, B., Maassen, J., Seinstra, F. J., & Bal, H. E. (2014, May). Performance models for CPU-GPU data transfers. In 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (pp. 11-20). IEEE.

40. CUDA C++ Best Practices Guide [Електронний ресурс]. NVIDIA Developer. Версія 13.3. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/ (дата звернення: 05.05.2026)

41. Wan, L., Li, K., Liu, J., & Li, K. (2016). Efficient CPU-GPU cooperative computing for solving the subset-sum problem. Concurrency and Computation: Practice and Experience, 28(2), 492-516.

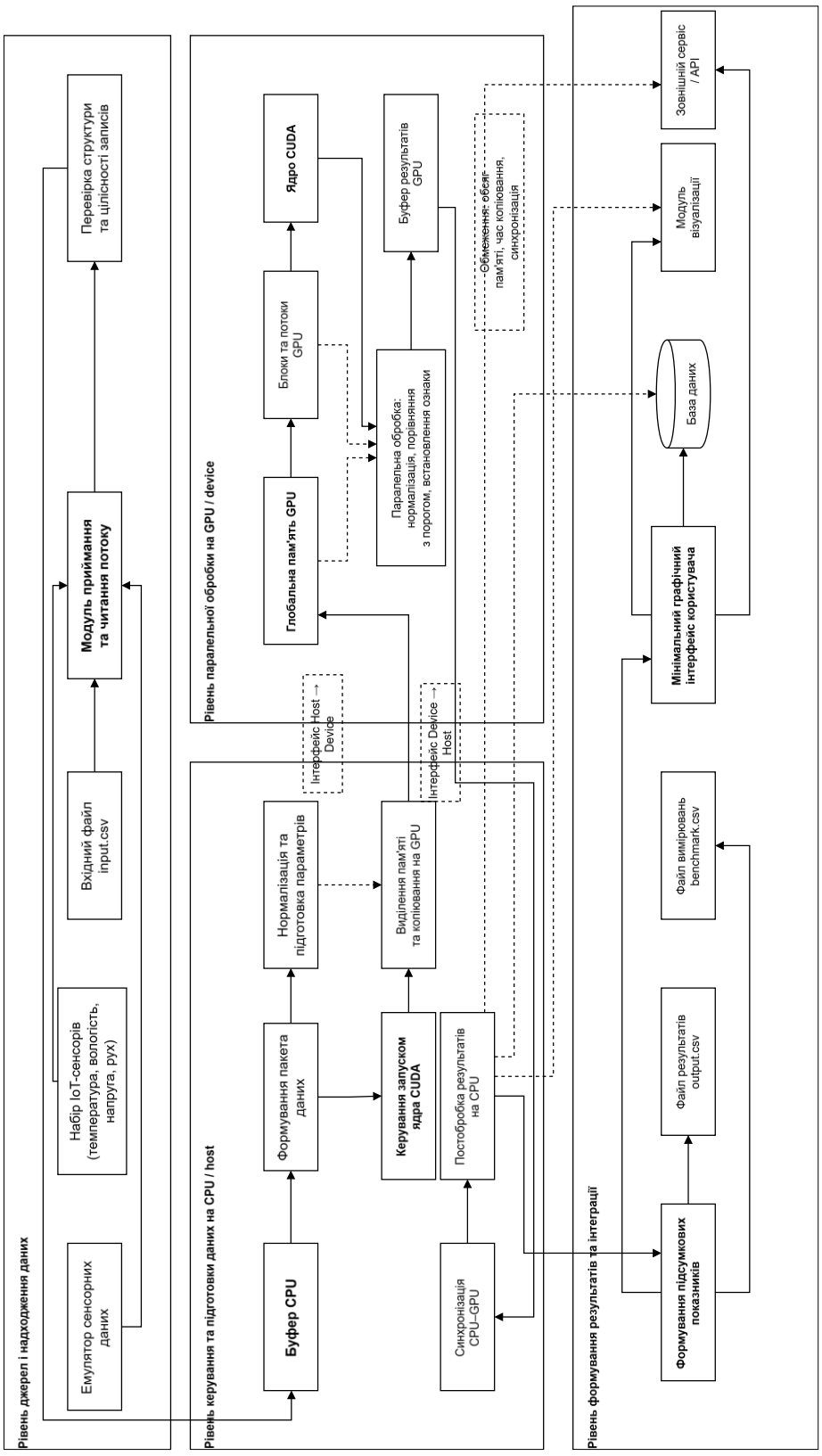
42. Комар М.П., Саченко А.О., Васильків Н.М., Гладій Г.М., Коваль В.С., Ліп’яніна-Гончаренко Х.В. Методичні рекомендації до виконання кваліфікаційної роботи з освітньо-професійної програми «Комп’ютерні науки» спеціальності 122 «Комп’ютерні науки» за першим (бакалаврським) рівнем вищої освіти. Тернопіль: ЗУНУ, 2024. 52 с.

43. Романюк Р. С. Програмний модуль паралельної обробки сенсорних потоків IoT на базі технології CUDA. Науковий простір: аналіз, сучасний стан,

тренди та перспективи : матеріали IX Всеукр. студент. наук. конф. (м. Київ, 19 черв. 2026 р.). Київ, 2026.

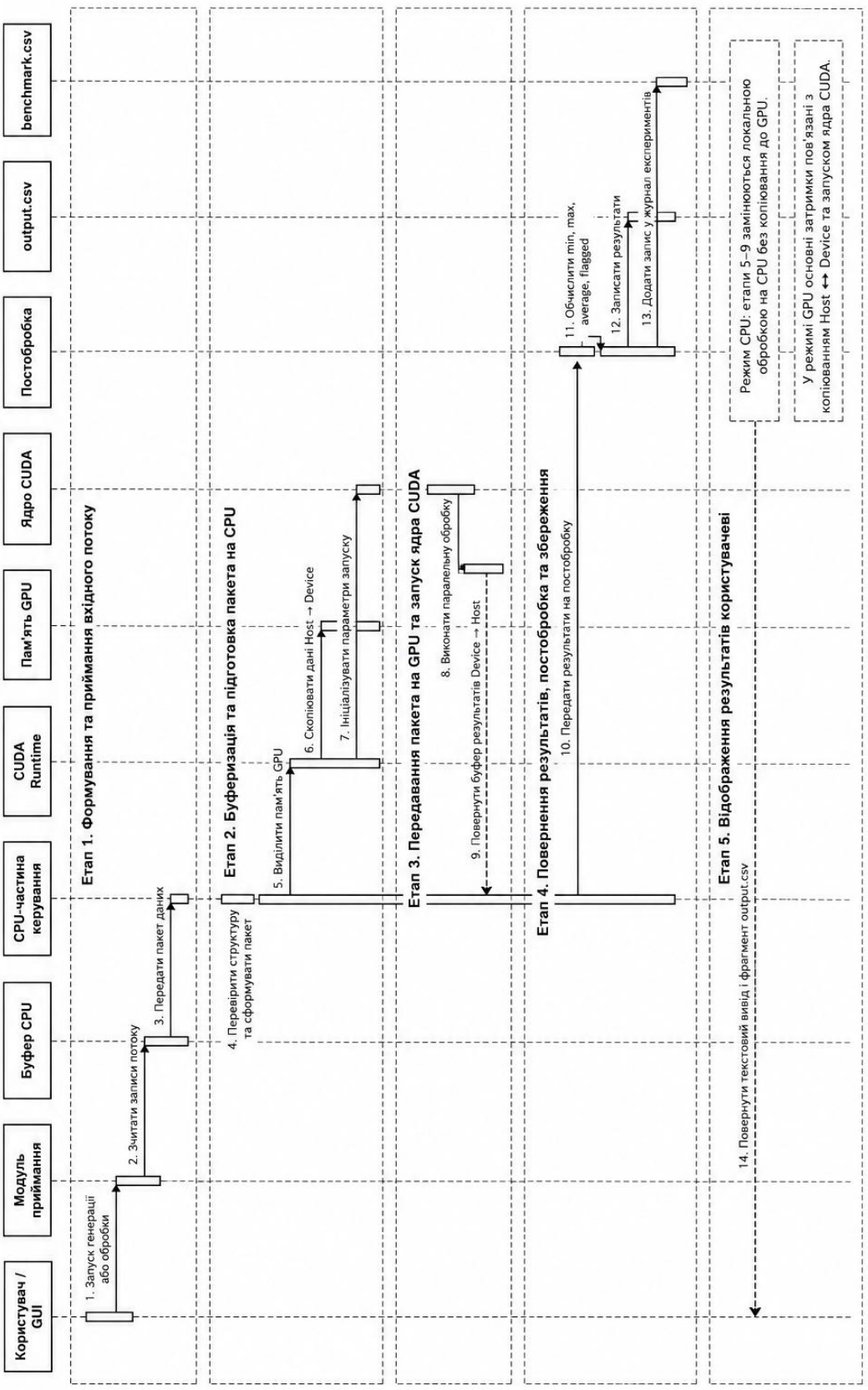
Додаток А

Розширена архітектура програмного модуля паралельної обробки сенсорних потоків IoT



Додаток Б

Діаграма послідовності обробки пакета сенсорних даних у взаємодії CPU та GPU



Додаток В

Код програмно апаратного модуля

```
#include <cuda_runtime.h>
#include <algorithm>
#include <chrono>
#include <cstdio>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <limits>
#include <sstream>
#include <string>
#include <vector>

struct SensorRecord {
    int id;
    float inputValue;
};

struct ProcessingStats {
    float minValue = 0.0f;
    float maxValue = 0.0f;
    float averageValue = 0.0f;
    int flaggedCount = 0;
    float flaggedPercent = 0.0f;
};

__global__ void processSensorDataGpu(const float* inputValues,
                                     float* normalizedValues,
                                     int* flags,
                                     int recordCount,
                                     float threshold) {
    const int index = blockIdx.x * blockDim.x + threadIdx.x;
    if (index >= recordCount) {
        return;
    }

    const float value = inputValues[index];
    normalizedValues[index] = value / 100.0f;
    flags[index] = (value >= threshold) ? 1 : 0;
}

bool readInputCsv(const std::string& filePath, std::vector<SensorRecord>& records) {
    std::ifstream inputFile(filePath);
    if (!inputFile.is_open()) {
        std::cerr << "Cannot open input file: " << filePath << std::endl;
        return false;
    }

    std::string line;
    if (!std::getline(inputFile, line)) {
        std::cerr << "Input file is empty." << std::endl;
        return false;
    }

    while (std::getline(inputFile, line)) {
        if (line.empty()) {
            continue;
        }

        std::stringstream lineStream(line);
        std::string idText;
        std::string valueText;

        if (!std::getline(lineStream, idText, ',')) {
            continue;
        }
        if (!std::getline(lineStream, valueText, ',')) {
            continue;
        }
```

```
        try {
            SensorRecord record{};
            record.id = std::stoi(idText);
            record.inputValue = std::stof(valueText);
            records.push_back(record);
        } catch (...) {
            std::cerr << "Skipped invalid line: " << line << std::endl;
        }
    }

    return !records.empty();
}

void processCpu(const std::vector<float>& inputValues,
                std::vector<float>& normalizedValues,
                std::vector<int>& flags,
                float threshold) {
    for (std::size_t i = 0; i < inputValues.size(); ++i) {
        normalizedValues[i] = inputValues[i] / 100.0f;
        flags[i] = (inputValues[i] >= threshold) ? 1 : 0;
    }
}

bool processGpu(const std::vector<float>& inputValues,
                std::vector<float>& normalizedValues,
                std::vector<int>& flags,
                float threshold) {
    const int recordCount = static_cast<int>(inputValues.size());

    float* deviceInput = nullptr;
    float* deviceNormalized = nullptr;
    int* deviceFlags = nullptr;

    cudaError_t status = cudaSuccess;

    status = cudaMalloc(reinterpret_cast<void**>(&deviceInput), recordCount * sizeof(float));
    if (status != cudaSuccess) {
        std::cerr << "cudaMalloc for input failed: " << cudaGetErrorString(status) << std::endl;
        return false;
    }

    status = cudaMalloc(reinterpret_cast<void**>(&deviceNormalized), recordCount * sizeof(float));
    if (status != cudaSuccess) {
        std::cerr << "cudaMalloc for normalized values failed: " << cudaGetErrorString(status) << std::endl;
        cudaFree(deviceInput);
        return false;
    }

    status = cudaMalloc(reinterpret_cast<void**>(&deviceFlags), recordCount * sizeof(int));
    if (status != cudaSuccess) {
        std::cerr << "cudaMalloc for flags failed: " << cudaGetErrorString(status) << std::endl;
        cudaFree(deviceInput);
        cudaFree(deviceNormalized);
        return false;
    }

    status = cudaMemcpy(deviceInput,
                        inputValues.data(),
                        recordCount * sizeof(float),
                        cudaMemcpyHostToDevice);
    if (status != cudaSuccess) {
        std::cerr << "cudaMemcpy HostToDevice failed: " << cudaGetErrorString(status) << std::endl;
        cudaFree(deviceInput);
        cudaFree(deviceNormalized);
        cudaFree(deviceFlags);
        return false;
    }

    const int threadsPerBlock = 256;
    const int blockCount = (recordCount + threadsPerBlock - 1) / threadsPerBlock;
    processSensorDataGpu<<<blockCount, threadsPerBlock>>>(deviceInput,
                                                          deviceNormalized,
```

```
                                                    deviceFlags,
                                                    recordCount,
                                                    threshold);

    status = cudaGetLastError();
    if (status != cudaSuccess) {
        std::cerr << "CUDA kernel launch failed: " << cudaGetErrorString(status) << std::endl;
        cudaFree(deviceInput);
        cudaFree(deviceNormalized);
        cudaFree(deviceFlags);
        return false;
    }

    status = cudaDeviceSynchronize();
    if (status != cudaSuccess) {
        std::cerr << "cudaDeviceSynchronize failed: " << cudaGetErrorString(status) << std::endl;
        cudaFree(deviceInput);
        cudaFree(deviceNormalized);
        cudaFree(deviceFlags);
        return false;
    }

    status = cudaMemcpy(normalizedValues.data(),
                        deviceNormalized,
                        recordCount * sizeof(float),
                        cudaMemcpyDeviceToHost);
    if (status != cudaSuccess) {
        std::cerr << "cudaMemcpy DeviceToHost for normalized values failed: "
                  << cudaGetErrorString(status) << std::endl;
        cudaFree(deviceInput);
        cudaFree(deviceNormalized);
        cudaFree(deviceFlags);
        return false;
    }

    status = cudaMemcpy(flags.data(),
                        deviceFlags,
                        recordCount * sizeof(int),
                        cudaMemcpyDeviceToHost);
    if (status != cudaSuccess) {
        std::cerr << "cudaMemcpy DeviceToHost for flags failed: "
                  << cudaGetErrorString(status) << std::endl;
        cudaFree(deviceInput);
        cudaFree(deviceNormalized);
        cudaFree(deviceFlags);
        return false;
    }

    cudaFree(deviceInput);
    cudaFree(deviceNormalized);
    cudaFree(deviceFlags);
    return true;
}

ProcessingStats calculateStats(const std::vector<float>& inputValues, const std::vector<int>& flags) {
    ProcessingStats stats{};
    if (inputValues.empty()) {
        return stats;
    }

    stats.minValue = std::numeric_limits<float>::max();
    stats.maxValue = std::numeric_limits<float>::lowest();
    double sum = 0.0;

    for (std::size_t i = 0; i < inputValues.size(); ++i) {
        stats.minValue = std::min(stats.minValue, inputValues[i]);
        stats.maxValue = std::max(stats.maxValue, inputValues[i]);
        sum += inputValues[i];
        stats.flaggedCount += flags[i];
    }

    stats.averageValue = static_cast<float>(sum / static_cast<double>(inputValues.size()));
    stats.flaggedPercent = static_cast<float>(stats.flaggedCount) * 100.0f /
                           static_cast<float>(inputValues.size());
```

```
    return stats;
}

bool writeOutputCsv(const std::string& filePath,
                    const std::vector<SensorRecord>& records,
                    const std::vector<float>& normalizedValues,
                    const std::vector<int>& flags) {
    std::ofstream outputFile(filePath);
    if (!outputFile.is_open()) {
        std::cerr << "Cannot open output file: " << filePath << std::endl;
        return false;
    }

    outputFile << "id,input_value,normalized_value,flag\n";
    for (std::size_t i = 0; i < records.size(); ++i) {
        outputFile << records[i].id << ','
                   << std::fixed << std::setprecision(2) << records[i].inputValue << ','
                   << std::fixed << std::setprecision(4) << normalizedValues[i] << ','
                   << flags[i] << '\n';
    }

    return true;
}

bool appendBenchmarkCsv(const std::string& filePath,
                        const std::string& mode,
                        int recordCount,
                        float threshold,
                        const ProcessingStats& stats,
                        double processingTimeMs) {
    const bool fileExists = std::filesystem::exists(filePath);
    std::ofstream benchmarkFile(filePath, std::ios::app);
    if (!benchmarkFile.is_open()) {
        std::cerr << "Cannot open benchmark file: " << filePath << std::endl;
        return false;
    }

    if (!fileExists) {
        benchmarkFile << "mode,record_count,threshold,min_value,max_value,average_value,flagged_count,"
                         "flagged_percent,processing_time_ms\n";
    }

    benchmarkFile << mode << ','
                  << recordCount << ','
                  << std::fixed << std::setprecision(2) << threshold << ','
                  << std::fixed << std::setprecision(2) << stats.minValue << ','
                  << std::fixed << std::setprecision(2) << stats.maxValue << ','
                  << std::fixed << std::setprecision(2) << stats.averageValue << ','
                  << stats.flaggedCount << ','
                  << std::fixed << std::setprecision(2) << stats.flaggedPercent << ','
                  << std::fixed << std::setprecision(3) << processingTimeMs << '\n';

    return true;
}

int main(int argc, char* argv[]) {
    const std::string inputFilePath = "input.csv";
    const std::string outputFilePath = "output.csv";
    const std::string benchmarkFilePath = "benchmark.csv";

    float threshold = 60.0f;
    std::string mode = "gpu";

    if (argc >= 2) {
        try {
            threshold = std::stof(argv[1]);
        } catch (...) {
            std::cerr << "Invalid threshold value." << std::endl;
            return 1;
        }
    }

    if (argc >= 3) {
```

```
        mode = argv[2];
        std::transform(mode.begin(), mode.end(), mode.begin(), [](unsigned char ch) {
            return static_cast<char>(std::tolower(ch));
        });
    }

    if (mode != "cpu" && mode != "gpu") {
        std::cerr << "Invalid mode. Use cpu or gpu." << std::endl;
        return 1;
    }

    std::vector<SensorRecord> records;
    if (!readInputCsv(inputFilePath, records)) {
        return 1;
    }

    std::vector<float> inputValues;
    inputValues.reserve(records.size());
    for (const auto& record : records) {
        inputValues.push_back(record.inputValue);
    }

    std::vector<float> normalizedValues(records.size(), 0.0f);
    std::vector<int> flags(records.size(), 0);

    const auto startTime = std::chrono::high_resolution_clock::now();

    bool success = false;
    if (mode == "cpu") {
        processCpu(inputValues, normalizedValues, flags, threshold);
        success = true;
    } else {
        success = processGpu(inputValues, normalizedValues, flags, threshold);
    }

    const auto endTime = std::chrono::high_resolution_clock::now();
    if (!success) {
        return 1;
    }

    const double processingTimeMs = std::chrono::duration<double, std::milli>(endTime - startTime).count();
    const ProcessingStats stats = calculateStats(inputValues, flags);

    if (!writeOutputCsv(outputFilePath, records, normalizedValues, flags)) {
        return 1;
    }

    if (!appendBenchmarkCsv(benchmarkFilePath,
                            mode,
                            static_cast<int>(records.size()),
                            threshold,
                            stats,
                            processingTimeMs)) {
        return 1;
    }

    std::cout << "Processing completed. Result written to output.csv" << std::endl;
    std::cout << "Mode: " << mode << std::endl;
    std::cout << "Record count: " << records.size() << std::endl;
    std::cout << std::fixed << std::setprecision(0) << "Threshold: " << threshold << std::endl;
    std::cout << std::fixed << std::setprecision(2) << "Minimum value: " << stats.minValue << std::endl;
    std::cout << std::fixed << std::setprecision(2) << "Maximum value: " << stats.maxValue << std::endl;
    std::cout << std::fixed << std::setprecision(2) << "Average value: " << stats.averageValue << std::endl;
    std::cout << "Flagged count: " << stats.flaggedCount << std::endl;
    std::cout << std::fixed << std::setprecision(2) << "Flagged percent: " << stats.flaggedPercent << std::endl;
    std::cout << std::fixed << std::setprecision(3) << "Processing time (ms): " << processingTimeMs <<
std::endl;

    return 0;
}
```

Додаток Г

Апробація отриманих результатів

Д О В І Д К А

ПРО ПРИЙНЯТТЯ ТЕЗ ДО ПУБЛІКАЦІЇ

10.06.2026

Шановний(і) авторе(и):
Романюк Ростислав Сергійович,

Організаційний комітет з радістю повідомляє, що тези «ПРОГРАМНИЙ МОДУЛЬ ПАРАЛЕЛЬНОЇ ОБРОБКИ СЕНСОРНИХ ПОТОКІВ IoT НА БАЗІ ТЕХНОЛОГІЇ CUDA» прийняті до публікації в збірнику за матеріалами IX Всеукраїнської студентської наукової конференції «Науковий простір: аналіз, сучасний стан, тренди та перспективи» (19.06.2026, м. Київ, Україна).

Опубліковані тези будуть доступні з 19.06.2026 за посиланням:
https://archive.liga.science/index.php/conference-proceedings/issue/view/ukr-19.06.2026

Електронні сертифікати учасників конференції та подяки науковим керівникам будуть доступні з 19 червня. Розсилка замовлених друкованих примірників, сертифікатів та подяк відбудеться до 3 липня.

Конференцію зареєстровано Державною науковою установою «УкрІНТЕІ» в базі даних науково-технічних заходів України та інформаційному бюлетені «План проведення наукових, науково-технічних заходів в Україні» (Посвідчення № 124 від 26.01.2026).

З повагою,

Директор Молодіжної наукової ліги
Голова оргкомітету конференції
ІГОР КОРЕНЮК

ПРОГРАМНИЙ МОДУЛЬ ПАРАЛЕЛЬНОЇ ОБРОБКИ СЕНСОРНИХ ПОТОКІВ IoT НА БАЗІ ТЕХНОЛОГІЇ CUDA

Романюк Ростислав Сергійович

Бакалавр спеціальності 122 – Комп'ютерні науки

Західноукраїнський національний університет, Україна

Науковий керівник: Осолінський Олександр Романович

к.т.н., доцент кафедри інформаційно-обчислювальних систем і управління

Західноукраїнський національний університет, Україна

Вступ

Сучасні системи Інтернету речей формують значні обсяги сенсорних даних, що надходять від фізичних пристроїв у вигляді часових послідовностей вимірювань. Такі потоки можуть містити показники температури, вологості, тиску, освітленості, концентрації газів або інших параметрів крім того поля можуть містити службову інформацію куди входить ідентифікатор пристрою, часова мітка та ознаки стану повідомлення. Через збільшення кількості сенсорів і частоти надходження даних традиційна послідовна обробка на центральному процесорі не завжди забезпечує потрібну швидкодію.

Тому є потреба в розробці програмного модуля паралельної обробки що зменшить час опрацювання великих наборів однотипних сенсорних записів. Технологія CUDA дає змогу перенести масові обчислювальні операції на графічний процесор, тоді як центральний процесор залишається відповідальним за приймання потоку.

Архітектура програмного модуля

Структура розробленого модуля охоплює джерела даних, приймання потоку, буфер пам'яті, керуючу частину на CPU, обчислювальні ядра CUDA на GPU, модуль формування результатів та інтерфейс взаємодії із зовнішніми компонентами. Такий поділ дозволяє відокремити логіку керування від масово-паралельної обробки та спростити подальше розширення системи.

На вхід модуля можуть надходити повідомлення від IoT-сенсорів, шлюзів або програмних емуляторів. Після первинної перевірки дані потрапляють в буфер, де накопичуються до формування пакета. CPU-частина відповідає за контроль буфера, підготовку параметрів запуску ядра CUDA, передачу даних між пам'яттю комп'ютера і пам'яттю GPU, синхронізацію та постобробку результатів. GPU-частина виконує однотипні операції над великою кількістю елементів: нормалізацію, порогову перевірку та формування проміжних результатів.

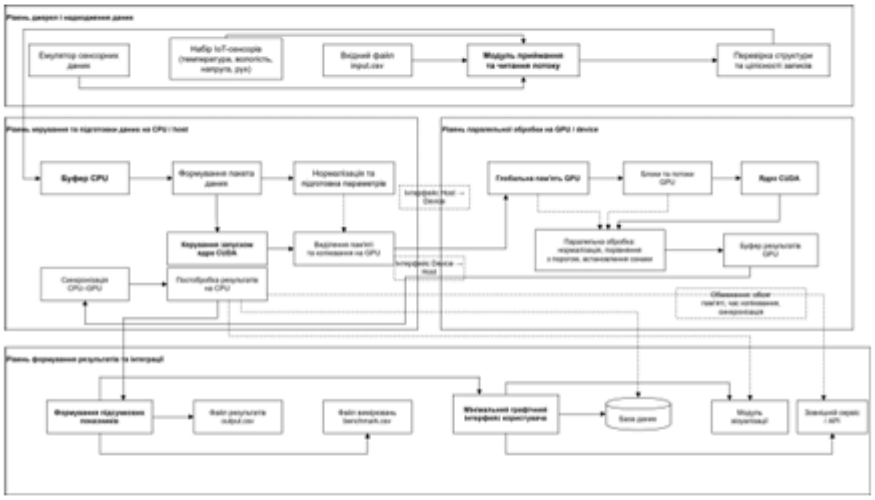


Рис. 1. **Архітектура програмного модуля паралельної обробки сенсорних потоків IoT**

Запропонована схема враховує можливі вузькі місця гетерогенної обробки. Найбільш критичними є витрати часу на копіювання даних між CPU та GPU, синхронізація після виконання ядра CUDA і вибір розміру пакета. Якщо пакет занадто малий, накладні витрати на передавання даних можуть перевищувати виграш від паралельного виконання. Якщо пакет надто великий, зростає навантаження на пам'ять GPU. Тому ефективність модуля визначається швидкодією ядра CUDA та від приймання потоку до збереження результатів.

Програмна реалізація та результати тестування

Основний модуль реалізовано мовою C++ із використанням CUDA C/C++. Він підтримує два режими виконання: послідовний режим cpu та паралельний режим gpu. Вхідні дані зчитуються з файлу csv, після чого кожний запис перетворюється у внутрішню структуру з ідентифікатором та числовим значенням. Для кожного елемента виконується нормалізація шляхом ділення значення на 100 та визначається прапорець перевищення заданого порогу.

У CPU-режимі обробка відбувається послідовним проходом по масиву. В GPU-режимі виконується виділення пам'яті на графічному процесорі, копіювання вхідних значень, запуск CUDA-ядра та повернення масивів результатів у пам'ять CPU. Після обробки програма формує статистику: мінімальне, максимальне та середнє значення, кількість записів, що перевищили поріг, відсоток таких записів і час виконання.

Для взаємодії з користувачем створено графічний інтерфейс мовою Python з використанням Tkinter. Інтерфейс дозволяє згенерувати вхідний набір даних, задати кількість записів, встановити порогове значення, вибрати режим CPU або GPU та запустити основний виконуваний модуль.

Рис. 2. **Результат експерименту для пакета з 1000000 записів**

Експериментальна перевірка виконувалася для наборів із 10, 100000 та 1000000 записів із пороговим значенням 60. Для малого набору з 10 записів CPU-режим показав час 0,000 мс, а GPU-режим - 0,420 мс, що пояснюється накладними витратами на копіювання і запуск обчислень. Для 100000 записів час CPU становив 1,472 мс, а GPU - 1,236 мс. Для 1000000 записів CPU виконав обробку за 14,880 мс, а GPU - за 9,605 мс. Отже, для великого пакета графічний процесор забезпечив прискорення приблизно у 1,55 раза, при цьому результати обробки в обох режимах збігалися.

Висновки. Було розроблено програмний модуль паралельної обробки сенсорних потоків IoT на базі технології CUDA, реалізовано обробку сенсорних записів в двох режимах, створено графічний інтерфейс для генерації даних і запуску експериментів. Тестування підтвердило коректність роботи модуля та показало, що використання GPU є доцільним для великих пакетів даних, тоді як для малих наборів перевага паралельної обробки нівелюється накладними витратами.

Список використаних джерел:

1. Krishnamurthi R., Kumar A., Gopinathan D., Nayyar A., Qureshi B. An Overview of IoT Sensor Data Processing, Fusion, and Analysis Techniques // *Sensors*. – 2020. – Vol. 20, No. 21. – Art. 6076.
2. ALHAFEZ, Nizar; KURDI, Ahmad. Parallel Paradigms in Modern HPC: A Comparative Analysis of MPI, OpenMP, and CUDA. *arXiv preprint arXiv:2506.15454, 2025.*
3. CUDA C++ Best Practices Guide [Електронний ресурс]. NVIDIA Developer. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html.