Міністерство освіти і науки України

Криворізький національний університет

Факультет інформаційних технологій

Кафедра комп’ютерних систем та мереж

Пояснювальна записка

до кваліфікаційної роботи бакалавра

за спеціальністю 123 «Комп’ютерна інженерія»

на тему: Комп’ютерна система з архітектурою на основі графічних процесорів типу CUDA.

|  |  |  |  |
| --- | --- | --- | --- |
| Проектував |  |  | А. І. Жижич |
| Керівник роботи |  |  | А. І. Купін |
| Нормоконтроль |  |  | Д. І. Кузнєцов |
| Завідувач кафедри |  |  | А. І. Купін |

Кривий Ріг

2025

Криворізький національний університет

Факультет інформаційних технологій

Кафедра комп’ютерних систем та мереж

Ступінь вищої освіти бакалавр

Спеціальність 123 «Комп’ютерна інженерія»

**ЗАТВЕРДЖУЮ**

Завідувач кафедри, голова циклової комісії

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ А. І. Купін

“\_\_\_\_” \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ 20\_\_ року

**ЗАВДАННЯ**

**НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТУ**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(прізвище, ім’я, по батькові)

1. Тема роботи \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

керівник роботи \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_,

(прізвище, ім’я, по батькові, науковий ступінь, вчене звання)

затверджені наказом вищого навчального закладу від “\_\_\_”\_\_\_\_\_\_ 20\_\_року №\_\_

2. Строк подання студентом роботи \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

3. Вихідні дані до роботи \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

5. Перелік графічного матеріалу (з точним зазначенням обов’язкових креслень)\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

6. Консультанти розділів роботи

|  |  |  |  |
| --- | --- | --- | --- |
| Розділ | Прізвище, ініціали та посада  консультанта | Підпис, дата | |
| завдання  видав | завдання  прийняв |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

7. Дата видачі завдання\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**КАЛЕНДАРНИЙ ПЛАН**

|  |  |  |  |
| --- | --- | --- | --- |
| №  з/п | Назва етапів роботи | Строк виконання етапів роботи | Примітка |
| 1 | Опис CUDA архітектури |  |  |
| 2 | Опис алгоритму обчислення |  |  |
| 3 | Проектування алгоритму обчислення суми |  |  |
| 4 | Вибір мови програмування |  |  |
| 5 | Розробка коду програми |  |  |
| 6 | Тестування розробленої програми та аналіз отриманих даних |  |  |
| 7 | Підготовка пояснювальної записки, презентації |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

**Студент** \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (прізвище та ініціали)

**Керівник роботи** \_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (прізвище та ініціали)

РЕФЕРАТ

Пояснювальна записка

Змн.

Арк.

№ документа

Підпис

Дата

Аркуш

КНУ.ПК. 123.24.18. АО

Розробив

Жижич

Купін

Перевірив

Н.контроль

Кузнєцов

Затвердив

Купін

РЕФЕРАТ

Літера

Аркушів

ЗКІ-21

Група: ЗКІ-21

Обсяг роботи: 49 сторінок, 10 ілюстрацій, 1 таблиця, 1 додаток, 10 використаних джерел.

Метою дослідження полягає у сворені програмного засобу для обчислення суми N випадкових чисел із застосуванням архітектури CUDA. У межах роботи розглянуто сучасні підходи до паралельних обчислень на графічних процесорах, що дають змогу значно підвищити продуктивність при роботі з великими обсягами даних.

Роботу огляу архітектури CUDA, розкрито її ключові принципи функціонування, а також проаналізовано основні переваги й обмеження, властиві цій технології. Зокрема, розглянуто структуру апаратного забезпечення, модель програмування, а також механізми управління потоками й пам’яттю, що є фундаментальними для ефективного використання CUDA.

ї програма яка дає змогу користувачу самостійно задавати кількість обчислювальних потоків для оптимізації роботи. Зокрема реалізації алгоритмів із урахуванням специфіки архітектури GPU. Описано основні функціональні модулі програми та обґрунтовано вибір структур даних і алгоритмічних рішень для досягнення максимальної продуктивності.

Результати тестування розробленої програми. Здійснено перевірку її коректності та стабільності за допомогою різноманітних тестових наборів даних, а також проведено детальний аналіз продуктивності. Аналіз базується на вимірюванні часу виконання, масштабованості алгоритмів та оцінці рівня паралелізму, що дозволяє визначити ефективність реалізації та можливі напрямки подальшого вдосконалення.

ABSTRACT

Explanatory Note

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК. 123.24.18.АО

Group: ZKI-21

Volume of work: 49 pages, 10 illustrations, 1 table, 1 appendix, 10 references.

The purpose of this research is to develop a software tool for calculating the sum of N random numbers using the CUDA architecture. The work explores modern approaches to parallel computing on graphics processing units (GPUs), which significantly enhance performance when working with large volumes of data.

The study provides an overview of the CUDA architecture, outlining its key operational principles and analyzing the main advantages and limitations inherent in this technology. Specifically, it examines the hardware structure, programming model, and mechanisms for thread and memory management, which are fundamental for efficient CUDA utilization.

The developed program allows users to manually specify the number of computational threads to optimize performance, taking into account the specifics of GPU architecture. The main functional modules of the program are described, and the choice of data structures and algorithmic solutions is justified to achieve maximum efficiency.

The results of testing the developed program are presented. The software was verified for correctness and stability using various test data sets, and a detailed performance analysis was conducted. This analysis is based on measuring execution time, algorithm scalability, and the level of parallelism, allowing for an assessment of implementation efficiency and the identification of possible directions for further improvement.

ЗМІСТ

Змн.

Арк.

№ документа

Підпис

Дата

Аркуш

КНУ.ПК.123.24.18.АО

Розробив

Жижич

Перевірив

Купін

Н.контроль

Кузнєцов

Затвердив

Купін

ЗМІСТ

Літера

Аркушів

ЗКІ-21

[РЕФЕРАТ 4](#_Toc199409209)

[ЗМІСТ 6](#_Toc199409210)

[ВСТУП 7](#_Toc199409211)

[1. ОЦІНЮВАННЯ АРХІТЕКТУРИ CUDA 10](#_Toc199409212)

[**1.1. Теоретичні аспекти архітектури CUDA** 10](#_Toc199409213)

[ОЦІНЮВАННЯ АРХІТЕКТУРИ CUDA 10](file:///C:\Users\nvsheludchenko\OneDrive%20-%20ArcelorMittal\Desktop\Бакалавр%20(CUDA).docx#_Toc199409214)

[Основні характеристики архітектури CUDA: 10](#_Toc199409215)

[Інструменти розробки у CUDA Toolkit: 11](#_Toc199409216)

[1.2 Теоретичні основи архітектури CUDA 12](#_Toc199409217)

[**1.3 Основні переваги архітектури CUDA:** 13](#_Toc199409218)

[Основні обмеження CUDA: 13](#_Toc199409219)

[Структура API CUDA 13](#_Toc199409220)

[1.4 Основи розробки програм на архітектурі CUDA 14](#_Toc199409221)

[1.5 Порівняння архітектур GPU та CPU 14](#_Toc199409222)

[15](#_Toc199409223)

[1.6. Програмна модель та модель пам’яті CUDA 16](#_Toc199409224)

[Програмна модель CUDA 16](#_Toc199409225)

[Модель пам’яті CUDA 17](#_Toc199409226)

[Висновки 18](#_Toc199409227)

[1.7 Проблеми використання технології CUDA 18](#_Toc199409228)

[1.8 Висновки за розділом 20](#_Toc199409229)

[2. МОДЕЛЬ ПРОГРАМУВАННЯ В CUDA 22](#_Toc199409230)

[**2.1 Основні поняття** 22](#_Toc199409231)

[Архітектурна модель CUDA 23](#_Toc199409232)

[Організація потоків у CUDA 25](#_Toc199409233)

[Етапи виконання CUDA-програми 26](#_Toc199409234)

[Основні засоб'и розробки 26](#_Toc199409235)

[**2.2.1 Специфікатори функцій й змінні** 27](#_Toc199409236)

Змн.

Арк.

№ документа

Підпис

Дата

Аркуш

КНУ.РБ.123.20.01.ВС

Розробив

Іванов

Перевірив

Купін

Н.контроль

Кузнєцов

Затвердив

Купін

ЗМІСТ

Літера

Аркушів

ЗКІ-21

[2.2.2 Спеціальні змінні в CUDA 28](#_Toc199409237)

[2.2.3 Синтаксис виклику ядра CUDA 29](#_Toc199409238)

[2.2.4 Розширені математичні функції в CUDA 29](#_Toc199409239)

[2.3 Висновки до розділу 31](#_Toc199409240)

[Таким чином, використання CUDA є доцільним для широкого спектра завдань, які вимагають інтенсивних обчислень, таких як машинне навчання, комп’ютерна графіка, обробка зображень, симуляції фізичних процесів тощо. 31](#_Toc199409241)

[3. ПРОЕКТУВАННЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ 32](#_Toc199409242)

[3.1. Обґрунтування вибору мови програмування та середовища розробки 32](#_Toc199409243)

[3.2. Розробка програмного забезпечення 34](#_Toc199409244)

[Підключення просторів Імен: 34](#_Toc199409245)

[3.3 Оцінка рівня паралелізму програми 40](#_Toc199409246)

[40](#_Toc199409247)

[**3.4** Висновки до розділу 42](#_Toc199409248)

[Висновки 43](#_Toc199409249)

[Розробка алгоритму та програмної реалізації 44](#_Toc199409250)

[Тестування та аналіз ефективності 44](#_Toc199409251)

[СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ 45](#_Toc199409252)

[ДОДАТОК А 46](#_Toc199409253)

ВСТУП

Ідея використання додаткових пристроїв для трансформації персональних комп’ютерів у міні-суперкомп’ютери з’явилась ще багато років тому. Ще у 1980-х роках на ринку були доступні спеціальні модулі — так звані трансп’ютери, які інтегрувалися в розширювальні слоти типу ISA, популярні на той час. Спершу такі рішення демонстрували значну продуктивність у певних типах завдань. Проте з часом класичні універсальні процесори значно пришвидшились, а також стали краще справлятися з паралельними обчисленнями. У результаті потреба в трансп’ютерах зникла. Водночас подібні спеціалізовані прискорювачі все ще використовуються, але зазвичай мають вузьке призначення та не здобули широкого розповсюдження.

Проте останніми роками паралельні обчислення активно впроваджуються в масовий сегмент, зокрема через індустрію тривимірних комп’ютерних ігор. Графічні адаптери з багатоядерною архітектурою, орієнтованою на векторні обчислення, що використовуються в 3D-графіці, забезпечують дуже високу продуктивність, недоступну традиційним CPU. Хоча максимальна ефективність досягається не у всіх задачах, подібні графічні процесори дедалі частіше застосовуються в областях, де раніше не планувалися. Яскравим прикладом такої архітектури є процесор Cell, створений спільно Sony, Toshiba та IBM і використаний у приставці PlayStation 3. До того ж, сучасні відеокарти від Nvidia та AMD також реалізують подібні обчислювальні підходи.

Сфера обчислень на графічних процесорах (GPU) зазнала стрімкого розвитку. Відповідно, провідні виробники відеочіпів — Nvidia та AMD — створили власні програмні платформи: CUDA (Compute Unified Device Architecture) та CTM (Close to Metal, або пізніше AMD Stream Computing). На відміну від старих підходів до програмування GPU, нові системи надають прямий доступ до апаратних ресурсів відеокарт. Хоча ці платформи несумісні між собою, CUDA є розширенням мови програмування C, тоді як CTM являє собою віртуальну машину, що виконує низькорівневий код.

Змн.

Арк.

№ документа

Підпис

Дата

Аркуш

КНУ.ПК. 123.24.18. АО

Розробив

*Жижич*

Купін

Перевірив

Н.контроль

Кузнєцов

Затвердив

Купін

ВСТУП

Літера

Аркушів

ЗКІ-21

Ці технології дозволили усунути значні обмеження попередніх моделей загальнопридатних обчислень на GPU, які базувались на графічних інтерфейсах, таких як Direct3D або OpenGL. Зокрема, Nvidia реалізувала платформу CUDA, яка включає спеціальну мову програмування, компілятор і бібліотеки, орієнтовані на виконання обчислень саме на графічному процесорі. Проте оптимізація такого коду є складним і трудомістким завданням, яке вимагає глибокого розуміння апаратної архітектури GPU. З іншого боку, це дає розробникам змогу максимально ефективно використовувати обчислювальні ресурси відеочіпів.

Платформа CUDA підтримується багатьма графічними процесорами Nvidia, зокрема серій G8x, G9x і GT2xx, які входять до складу популярних відеокарт GeForce поколінь 8, 9 і 200. На момент виходу фінальної версії CUDA 2.0 було реалізовано підтримку нових функцій, зокрема розрахунків з подвійною точністю. Платформа сумісна з 32- та 64-бітними версіями операційних систем Linux, Windows і macOS.

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК. 123.24.18.АО

1. ОЦІНЮВАННЯ АРХІТЕКТУРИ CUDA

**1.1. Теоретичні аспекти архітектури CUDA**

Змн.

Арк.

№ документа

Підпис

Дата

Аркуш

КНУ.ПК. 123.24.18. АО

Розробив

Жижич

Купін

Перевірив

Н.контроль

Кузнєцов

Затвердив

Купін

ОЦІНЮВАННЯ АРХІТЕКТУРИ CUDA

Літера

Аркушів

ЗКІ-21

У сфері високопродуктивних обчислень важливу роль відіграє платформа CUDA (Compute Unified Device Architecture), створена компанією NVIDIA як поєднання апаратного та програмного забезпечення.Вона забезпечує інструменти для створення програм, що дозволяють використовувати графічні процесори (GPU) не лише для графічного рендерингу, а й для вирішення завдань загального характеру. Основу цієї технології становить підхід GPGPU (обчислення загального призначення на графічних процесорах), який полягає у передачі частини обчислень від центрального процесора на GPU з метою підвищення ефективності обробки даних .

На відміну від попередніх підходів, що базувалися на використанні графічних API (наприклад, OpenGL, Direct3D), де обчислення реалізовувалися через шейдери, CUDA пропонує більш гнучке та зручне середовище. Програмування ведеться на мовах C або C++ із залученням спеціалізованих директив, що вказують на фрагменти коду, які мають виконуватись паралельно. Це дозволяє розробникам ефективніше реалізовувати обчислювальні алгоритми з високим ступенем паралелізму[2].

### **Основні характеристики архітектури CUDA:**

1. Уніфікована платформа для виконання паралельних обчислень на графічних процесорах NVIDIA.
2. Підтримка мов програмування C/C++ зі спеціальними розширеннями.
3. Вбудовані високопродуктивні бібліотеки для чисельних методів, такі як FFT (швидке перетворення Фур’є) і BLAS (базові операції лінійної алгебри).
4. Обмін данними між процесором і графічним процемором з мінімальними затримками.
5. Інтеграція з графічними інтерфейсами OpenGL та DirectX.
6. Підтримка різних операційних систем, включно з Windows, Linux і macOS, у 32- та 64-бітних версіях.
7. Можливість низькорівневої оптимізації обчислювальних процесів [3].

### **Інструменти розробки у CUDA Toolkit:**

* Компілятор nvcc;
* Бібліотеки для обчислень (cuFFT, cuBLAS);
* Інструменти профілювання та відлагодження (cuda-gdb);
* Документація, приклади коду та утиліти у складі CUDA SDK;
* Runtime-драйвери у стандартних драйверах NVIDIA.

Варто зауважити, що офіційна підтримка забезпечується переважно для провідних дистрибутивів Linux, таких як Red Hat і SUSE. Водночас, за відгуками спільноти CUDA, платформа демонструє стабільну роботу і на інших системах, включаючи Fedora, Ubuntu, Gentoo тощо.

Через важливість задач, пов’язаних із обробкою великих масивів даних (наприклад, у сфері логістики,медичне моделювання, фінансовий аналіз), зростає потреба у високопродуктивних системах. Використання GPU в таких системах є економічно обґрунтованим, оскільки дозволяє досягти високої продуктивності при нижчих витратах енергії та ресурсів на технічне обслуговування[4].

Архітектура CUDA стала основою для розвитку прикладних рішень у багатьох галузях:

* У наукових дослідженнях (наприклад, молекулярна динаміка в програмі AMBER, яка використовується понад 60 тисячами дослідників);
* У фінансових додатках (наприклад, компанії Numerix та CompatibL реалізували моделі аналізу фінансових ризиків, що прискорили обчислення в 18 разів);
* У промисловості — понад 700 обчислювальних кластерів з GPU Tesla встановлено у банківській, енергетичній та науковій сферах [5].

З розвитком операційних систем Windows 7 та macOS Snow Leopard, GPU перетворився з графічного пристрою на повноцінний співпроцесор для універсальних паралельних обчислень. Таким чином, CUDA стала ключовим інструментом реалізації паралельних рішень у масових та спеціалізованих продуктах.

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

Арк.

№ документа

Підпис

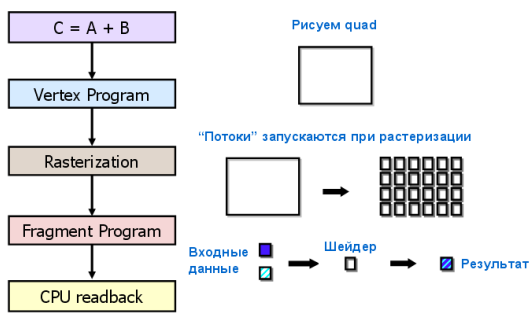
Дата

Арк.

КНУ.ПК.123.24.18.АО

## 1.2 Теоретичні основи архітектури CUDA

З погляду програміста, графічний конвеєр можна уявити як послідовність етапів обробки графічних даних. На етапі геометричної трансформації формується базова сцена у вигляді трикутників, тоді як на етапі растеризації відбувається перетворення цих трикутників у пікселі для подальшого виводу на екран В рамках класичної концепції GPGPU (виконання загального призначення обчислень на графічних процесорах) навіть найпростіші арифметичні операції таких як додавання двох векторів, потрібно формувати зображення або виводити результати у позаекранний буфер.



Малюнок 1.1 – програмування GPGPU

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

Цей підхід передбачає використання піксельних шейдерів, які реалізують обчислення кольору кожного пікселя на основі його координат. Для цього необхідно зчитувати вхідні дані з текстур, виконувати арифметичні дії над ними та зберігати результат у буфері. Такі операції, що в традиційному програмуванні виконуються єдиною командою, у GPGPU-виконанні вимагають значного програмного забезпечення.

Класичний GPGPU-підхід був обмежений складністю навчання, низькою гнучкістю мов шейдерів, обмеженнями щодо доступу до пам’яті та відсутністю універсальності. Дані представлялись у текстурному вигляді, а програми — у вигляді процедур рендерингу, що створювало суттєві обмеження для широкого використання.

Революційним кроком стало впровадження архітектури CUDA NVIDIA, яка дозволила реалізовувати обчислення загального призначення мови програмування C з мінімальними розширеннями. CUDA працює незалежно від графічних API (таких як OpenGL або DirectX), забезпечуючи прямий доступ до обчислювальних ресурсів GPU .

### **1.3 Основні переваги архітектури CUDA:**

* Інтеграція з мовою C: програмування здійснюється на знайомій мові з підтримкою покажчиків та циклів, що значно спрощує розробку.
* Shared memory: кожен мультипроцесор має локальну пам’ять обсягом 16 КБ, доступну для всіх потоків у блоці, що дозволяє реалізовувати кеші з високою пропускною здатністю.
* Покращена передача даних між оперативною пам’яттю CPU та пам’яттю GPU.
* Лінійна адресація та підтримка розширених операцій, включно з gather, scatter та записами у довільні адреси.
* Підтримка операцій над цілими числами та бітових маніпуляцій на рівні апаратури.

### **Основні обмеження CUDA:**

* Відсутність рекурсії у виконуваних функціях.
* Ширина блоку обчислень повинна бути кратною 32 (розмір warp).
* Пропрієтарність архітектури — CUDA є власністю NVIDIA і не підтримується на GPU інших виробників.

### **Структура API CUDA**

CUDA надає два основні API:

* CUDA Runtime API — це високорівневе програмне середовище, що спрощує розробку додатків завдяки автоматизованому управлінню обчислювальними ресурсами.
* CUDA Driver API — низькорівнева бібліотека, яка дозволяє програмісту більш точно керувати ресурсами та поведінкою пристрою.

Знати треба, що обидва API не можуть використовуватися одночасно в одній програмі. Високорівневий API фактично є надбудовою над драйверним рівнем, забезпечуючи простіший інтерфейс для програмування .

Таким чином, CUDA забезпечує ефективну платформу для створення високопродуктивних обчислювальних додатків, даючи змогу розробникам максимально ефективно використовувати всі переваги паралельної обробки даних на сучасних графічних процесорах

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

### **1.4 Основи розробки програм на архітектурі CUDA**

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

Графічний процесор (GPU) компанії NVIDIA базується на модульній архітектурі, що складається з декількох кластерів обробки текстур , які включають текстурні модулі, а також кілька потокових мультипроцесорів. Кожен SM містить багато ядер, які функціонують за схемою SIMD (Single Instruction, Multiple Data), тобто виконують одну інструкцію над різними наборами даних.

При роботі з великими обсягами даних — наприклад, при обробці зображень або фізичному моделюванні — запускаються тисячі або мільйони потоків, які групуються в warp’и (по 32 потоки в кожному). Один мультипроцесор може одночасно обслуговувати до 32 таких груп, що забезпечує масивну паралельність (NVIDIA, 2021).

Початковим етапом перенесення обчислювальної задачі на GPU є профілювання — аналіз програмного коду для виявлення “вузьких місць” продуктивності. Компіляція CUDA-програм відбувається за допомогою компілятора nvcc, який генерує машинний код як для центрального процесора (CPU), так і для графічного (GPU). Під час виконання CPU керує загальним потоком виконання, бере на себе ресурсомісткі паралельні обчислення (Sanders & Kandrot, 2010).

### **1.5 Порівняння архітектур СPU та GPU**

Через фізичні обмеження щодо частоти тактового сигналу, а також через енергоспоживання, сучасна обчислювальна індустрія поступово відходить від нарощування частоти CPU на користь мультикомпонентних рішень. Так, більшість сучасних центральних процесорів мають до 8 ядер, кожне з яких здатне виконувати окремий потік інструкцій.

У відповідь на потребу в швидшій обробці мультимедійних даних до CPU почали додаватися векторні розширення — SSE, SSE2, AVX тощо. Однак GPU, на відміну від CPU, з самого початку створені для високоефективного паралельного опрацювання великих масивів однотипних даних.

Наприклад, мультипроцесор графічного адаптера NVIDIA може містити десятки або сотні ядер, кожне , обмеженої спільної пам’яті (shared memory) та глобальної пам’яті, доступної для всіх обчислювальних блоків GPU. Ця архітектура дозволяє ефективно реалізовувати масово-паралельні алгоритми, особливо в задачах, де всі потоки виконують схожі інструкції — саме це і є ключовою перевагою архітектури типу SIMD або SIMT (NVIDIA, 2020).

Арк.

№ документа

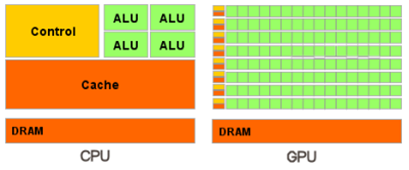
Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

На відміну від CPU, які орієнтовані на складну логіку керування потоками, кешування та спекулятивне виконання, GPU призначені для максимальної арифметичної щільності. Наприклад, архітектура Tesla дозволяє поєднання інструкцій, таких як MAD (Multiply-Add) з MUL або SFU (Special Function Unit) (Kirk & Hwu, 2016).



малюнок 1.1 - Эність пам’яті яку займає CPU та GPU

Щодо доступу до пам’яті — CPU працює з непередбачуваними (рандомними) шаблонами звернення, що потребує великих кешів і логіки передбачення. У GPU ж потоки зазвичай звертаються до пам’яті упорядковано, що знижує вимоги до обсягу кеш-пам’яті.

CPU значну частину площі кристала витрачає на логіку керування, тоді як у GPU основна частка припадає на виконавчі блоки та обчислювальні модулі. Один мультипроцесор у GPU може підтримувати до 1024 активних потоків і швидко перемикатися між ними, в силу чому досягається висока ефективність навіть при наявності латентності пам’яті.

Таким чином, хоча CPU краще підходять для послідовних, умовно складних обчислень із нерегулярною структурою доступу до пам’яті, GPU виграють у задачах масового паралелізму з однаковими типами обчислень. Прикладом можуть бути фізичне моделювання, обробка зображень, машинне навчання, моделювання біомолекул тощо. Особливо ефективним є використання GPU у задачах, де співвідношення між кількістю арифметичних операцій і зверненнями до пам’яті є високим.

Арк.

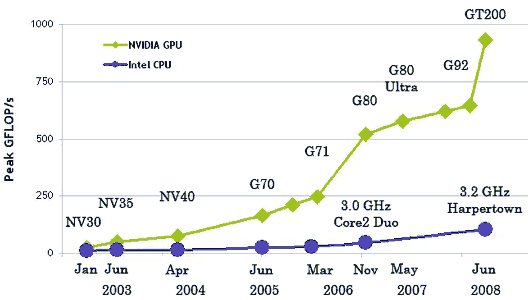
№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО



малюнок 1.2 – Зростання продуктивності

На рис. 1.2 показано графік зростання теоретичної продуктивності CPU та GPU, наведений компанією NVIDIA. Важливо враховувати, що вказані дані можуть бути оптимістичними щодо GPU, оскільки наведено результати для обчислень з одинарною точністю, яка не завжди відповідає потребам наукових задач. Проте, навіть з урахуванням цієї поправки, для багатьох типів паралельних обчислень GPU демонструють значну перевагу над CPU.

## 1.6. Програмна модель та модель пам’яті CUDA

У контексті паралельного програмування платформа CUDA, компанією NVIDIA розроблена, забезпечує доступ до графічного процесора (GPU) як до високопродуктивного обчислювального ресурсу, що функціонує як співпроцесор основної системи — центрального процесора (CPU або host). GPU має власну ієрархічну архітектуру обчислювальних блоків і пам’яті, що дозволяє ефективно обробляти тисячі потоків одночасно.

### **Програмна модель CUDA**

1. Програмна модель CUDA базується на запуску спеціальних функцій, які називаються ядрами (kernels), на GPU. Ці функції виконуються паралельно великою кількістю потоків (threads), кожен з яких працює незалежно. Потоки формують блоки, які в подальшому впорядковуються у сітку, складену з цих блоків.

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

Блоки можуть мати одно-, дво- або тривимірну структуру і зазвичай містять від 64 до 1024 потоків. Потоки в межах одного блоку можуть спільно використовувати пам’ять, синхронізуватись і обмінюватися даними. Потоки в блоці організовані у warp’и — групи по 32 потоки, що є базовою одиницею виконання на апаратному рівні.

Запуск ядра виконується над усією сіткою, і лише одна сітка може бути активною протягом одного виклику ядра. Якщо ресурси GPU дозволяють, блоки можуть виконуватися паралельно, інакше вони плануються на виконання послідовно. Такий підхід дозволяє масштабувати програми до великих обсягів обчислень без втрати ефективності.

### **Модель пам’яті CUDA**

Для написання ефективних програм на CUDA програміст має враховувати особливості моделі пам’яті GPU, оскільки правильний вибір типу пам’яті прямо впливає на продуктивність. У CUDA визначено декілька видів пам’яті з різною швидкодією, розміром і можливістю доступу:

1. Глобальна пам’ять (Global memory) — доступна для всіх потоків і має найбільший обсяг, але і найвищу затримку. Обмін інформацією між центральним та графічним процесорами здійснюється через функцію cudaMemcpy.

* Через високу латентність рекомендується мінімізувати доступ до неї.
* Регістри (Register memory) — найшвидший тип пам’яті, доступний лише одному потоку. Використовується для зберігання простих змінних. Їх обмежена кількість вимагає раціонального використання.
* Локальна пам’ять (Local memory) — використовується у випадках, коли регістри перевантажені. Незважаючи на назву, розміщується у глобальній пам’яті, тому доступ до неї повільний. Рекомендується уникати її надмірного використання.
* Shared memory — це високошвидкісна пам’ять, доступна для спільного використання потоками всередині одного блоку.
* Дає змогу ефективно обмінюватися даними та синхронізувати обчислення в межах блоку. Задається специфікатором \_\_shared\_\_.

1. Константна пам’ять (Constant memory) — кешована пам’ять малого обсягу (до 64 КБ), призначена для зберігання значень, що не змінюються під час виконання ядра. Передача даних з CPU до GPU у потоки відбувається за допомогою функції cudaMemcpyToSymbol.

* Текстурна пам’ять (Texture memory) — оптимізована для читання двовимірних масивів даних, зокрема текстур. Дозволяє ефективно працювати з кешованими зразками зображень або матриць. Для доступу до неї використовуються спеціальні API-функції, такі як cudaBindTexture().

Арк.

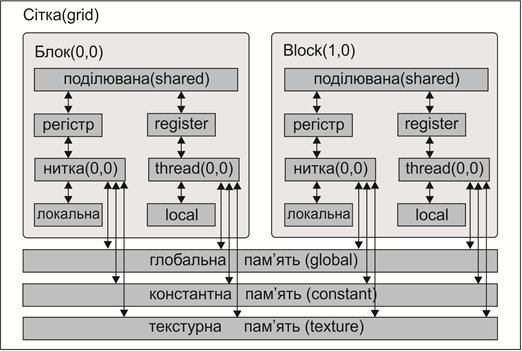
№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО



малюнок 1.3 – Глобальна пам’ять

### **Висновки**

Архітектура CUDA забезпечує розробнику високий рівень контролю над розподілом обчислювальних задач та доступом до різних типів пам’яті. Оптимальне поєднання програмної моделі та моделі пам’яті дозволяє максимально ефективно використовувати ресурси GPU. Знання особливостей пам’яті — один із ключових чинників написання продуктивних програм у середовищі CUDA.

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

### **1.7 Проблеми використання технології CUDA**

На сьогоднішній день архітектура CUDA активно застосовується як у наукових, так і в комерційних проєктах, охоплюючи сфери машинного навчання, обробки зображень, симуляцій фізичних процесів та багато інших. Детальні приклади таких реалізацій представлені на офіційному сайті NVIDIA [1]. Незважаючи на численні переваги, ця технологія має низку обмежень і проблем, які стримують її ще ширше впровадження.

Передусім, багато задач, особливо ті, що не потребують високої паралельності або тісно інтегровані в традиційні програмні стекові рішення, ефективно виконуються на центральних процесорах без потреби в GPGPU-прискоренні. Крім того, значна частина складних обчислень у комп’ютерній графіці вже давно делегована спеціалізованим графічним процесорам (3D-акселераторам), які виконують функції рендерингу незалежно від CUDA. Таким чином, частина навантаження, яка могла б оброблятись за допомогою CUDA, вже закріплена за іншими спеціалізованими апаратними рішеннями .

Ще одним викликом є питання сумісності. Багато користувачів, зокрема в корпоративному середовищі, віддають перевагу рішенням, які забезпечують кросплатформенність та просту адаптацію існуючого коду. Відсутність прямої підтримки CUDA з боку інших виробників, таких як AMD, створює бар’єри для універсалізації цієї технології. Якби, наприклад, AMD реалізувала сумісність із CUDA, навіть із нижчим рівнем продуктивності, це могло б стимулювати ширше інвестування в розробку відповідного ПЗ .

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

1. Компанія AMD, натомість, обрала шлях підтримки відкритого стандарту OpenCL — універсального фреймворку для гетерогенних обчислень. Попри схожість функціоналу з CUDA, різниця в API, синтаксисі та принципах оптимізації створює додаткові труднощі при адаптації рішень між платформами.
2. Ще один аспект, що обмежує широке впровадження CUDA, пов’язаний із високоспеціалізованими програмними рішеннями. У багатьох підприємствах ПЗ розробляється під конкретне апаратне забезпечення і не призначене для масового використання. Це обмежує потенціал масштабування, а також підвищує вартість інтеграції нових технологій. У країнах із високими витратами на оплату праці часто виявляється економічно доцільнішим придбати потужніші CPU, ніж наймати додаткових фахівців для оптимізації під CUDA. У країнах з обмеженими фінансами на розробку, де основний акцент ставиться на мінімізацію витрат, використання CUDA часто є більш доцільним варіантом.

Ще одна проблема — обмеження сумісності CUDA з деякими версіями операційних систем. Наприклад, Windows, не будучи системою реального часу, іноді перериває виконання CUDA-коду, що триває понад кілька секунд. Це може призводити до зависання або непередбачуваної поведінки програм, змушуючи розробників застосовувати нестандартні методи, як-от використання таймерів або зміну налаштувань драйверів .

Також слід відзначити технічне обмеження, пов’язане з низькою ефективністю обробки чисел подвійної точності (double precision). У багатьох наукових галузях саме формат double є основним, однак графічні процесори традиційно орієнтовані на обчислення з одинарною точністю (float), що значно продуктивніше. Такий фокус обумовлений історичним розвитком GPU як пристроїв для рендерингу, де подвійна точність майже не використовується .

Разом із тим, нові покоління GPU, зокрема архітектура Fermi, вже демонструють рівнозначну підтримку double і float, що суттєво розширює сферу застосування CUDA за межі графіки. У цій архітектурі також впроваджено численні покращення — від збільшеного об’єму кешу до оптимізованої роботи з пам’яттю, — які дозволяють говорити про CUDA як про потенційно революційну платформу для високопродуктивних обчислень.

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО



### **1.8 Висновки за розділом**

У 2007 році компанія NVIDIA представила інноваційну технологію— програмно-апаратну платформу, яка відкрила нові можливості у сфері високопродуктивних обчислень. Основною метою розробки стало забезпечення доступу до паралельного обчислювального потенціалу графічних процесорів (GPU), що дозволяє суттєво підвищити продуктивність персональних комп’ютерів і серверів (NVIDIA, 2007).

Архітектура CUDA дає змогу програмістам напряму працювати з ядрами GPU для вирішення обчислювально інтенсивних задач. Вона реалізує модель масово-паралельного програмування, в якій тисячі потоків можуть виконуватись одночасно, що значно перевищує рівень паралелізму, доступний для традиційних центральних процесорів (CPU). Це дозволяє скоротити час виконання складних математичних та інженерних розрахунків у декілька разів.

Платформа підтримує декілька мов програмування, зокрема C, C++, Python, Fortran, Java, що робить її зручною для використання широким колом розробників — від науковців до інженерів та спеціалістів з обробки даних. CUDA використовує гнучку ієрархічну модель організації обчислень: grid → block → thread, яка забезпечує ефективний розподіл задач між потоками та оптимізацію використання ресурсів GPU.

На сьогодні технологія CUDA широко застосовується у найрізноманітніших галузях. Серед них:

* наукові дослідження, що потребують моделювання фізичних процесів;
* штучний інтелект та машинне навчання, де GPU використовуються для тренування нейронних мереж;
* обробка зображень та відео в реальному часі;
* медична візуалізація, де велике значення має швидкість обробки томографічних даних;

Таким чином, технологія CUDA відіграє ключову роль у розвитку сучасних обчислювальних систем, забезпечуючи гнучкий, масштабований і потужний інструментарій для реалізації задач, які раніше вважалися надто ресурсоємними для стандартних процесорів. Вона є фундаментальною платформою для майбутніх інновацій у сфері паралельних обчислень.

Арк.

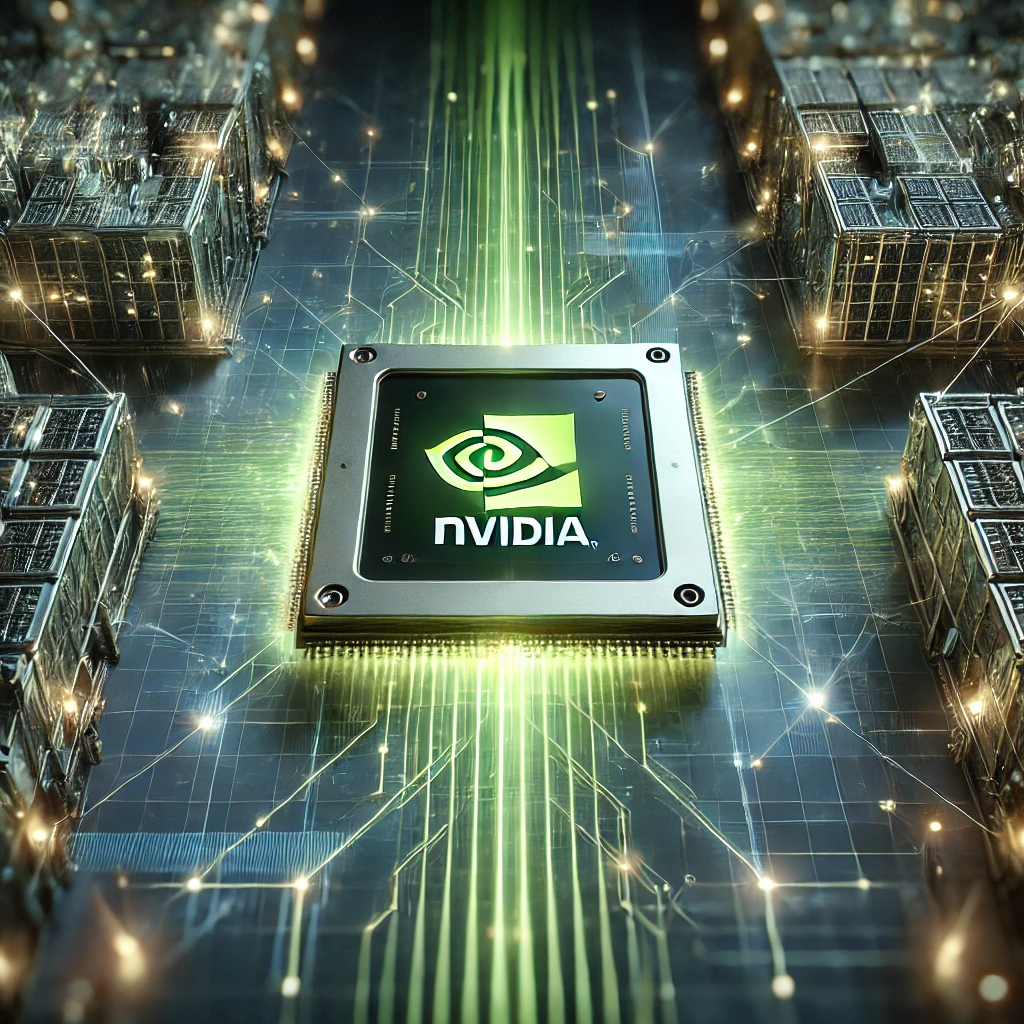
№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО



**2. МОДЕЛЬ ПРОГРАМУВАННЯ В CUDA**

Змн.

Арк.

№ документа

Підпис

Дата

Аркуш

КНУ.РБ.123.20.01.ВС

Розробив

Жижич

Перевірив

Купін

Н.контроль

Кузнєцов

Затвердив

Купін

МОДЕЛЬ ПРОГРАМУВАННЯ

Літера

Аркушів

ЗКІ-21

**2.1 Основні поняття**

CUDA це технологія, розроблена компанією NVIDIA, яка суттєво полегшує створення застосунків із використанням загальнопризначених обчислень на графічних процесорах (GPGPU). На відміну від традиційних графічних API (таких як OpenGL або DirectX), CUDA не має їх обмежень і дозволяє повноцінно використовувати обчислювальні можливості GPU для не графічних завдань [1].

Основне призначення CUDA — забезпечити ефективну підтримку масово-паралельних обчислень. Вона сумісна з усіма графічними адаптерами NVIDIA, починаючи з серії GeForce 8, а також з професійними обчислювальними рішеннями, такими як Tesla. CUDA стала стандартом у галузі наукових обчислень, машинного навчання, обробки зображень та моделювання складних фізичних процесів.

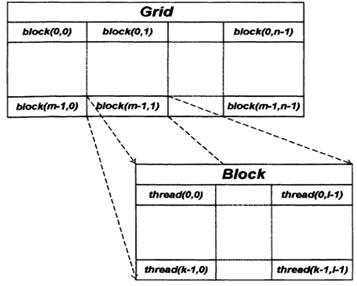
Однією з ключових переваг технології є її доступність: розробницький комплект (SDK), приклади та документація безкоштовно надаються на офіційному порталі [developer.nvidia.com](https://urldefense.com/v3/__https:/developer.nvidia.com__;!!AB_04_y_3-SRqw!8TmqH9NWL9icqfYVVmOYXRZI4w0IDmbVdbvt_Pkf7mcyY3mxe5LkuJbPWylgSa8bca_8-Z1jhhPva0N7bJNQjDgYPPZ0RQ$). Актуальні версії CUDA підтримують всі популярні операційні системи: Windows, Linux та macOS.

### **Архітектурна модель CUDA**

Архітектура CUDA базується на моделі гетерогенних обчислень, де центральний процесор (CPU) виконує роль хоста (host), а графічний процесор (GPU) — пристрою (device). Програма, створена з використанням CUDA, виконує послідовні частини алгоритму на CPU, а паралельні — на GPU, використовуючи множину одночасно виконуваних потоків (threads) [2].

GPU у цій моделі розглядається як паралельний співпроцесор, що:

* має власну виділену пам’ять;
* підтримує паралельне виконання тисяч потоків;
* працює під управлінням CPU, що ініціює виконання завдань.



Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

малюнок 2.1 – Нитки ієрархії

На відміну від традиційних потоків CPU, потоки GPU:

* створюються та керуються з мінімальними витратами;
* мають компактний контекст виконання (наприклад, регістри та стек);
* ефективно масштабуються до десятків тисяч одночасно активних потоків.

CUDA дозволяє писати програми мовою C/C++, доповненою спеціальними конструкціями, такими як специфікатори \_\_global\_\_, \_\_device\_\_, \_\_host\_\_, а також директивами для запуску ядер (kernel launch) та вбудованими змінними (threadIdx, blockIdx, blockDim, gridDim) [3].

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

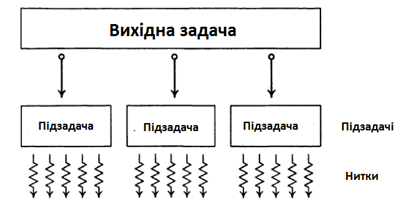


Рисунок 2.2 – Схема задачі

### **Організація потоків у CUDA**

Програмна модель CUDA передбачає ієрархічну організацію потоків:

* Grid — сітка, що складається з блоків (blocks) і представляє собою одновимірний або двовимірний масив блоків;
* Block — блок, який містить множину потоків (threads) і може бути одно-, дво- або тривимірним;
* Thread — окремий потік виконання, який обробляє одну або кілька одиниць даних.

Кожен потік має унікальний ідентифікатор в межах блоку (threadIdx) і кожен блок — у межах сітки (blockIdx). Це дозволяє ефективно ділити обчислювальні задачі між потоками.

Арк.

№ документа

Підпис

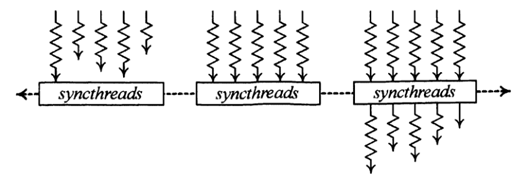
Дата

Арк.

КНУ.ПК.123.24.18.АО

Для визначення розмірів блоку та сітки використовуються змінні blockDim і gridDim, що дозволяє масштабувати обчислення в залежності від задачі. Потоки в межах одного блоку можуть взаємодіяти між собою через shared memory — швидку спільну пам’ять, яка фізично розміщується ближче до обчислювальних ядер GPU.

Оскільки не всі потоки фізично виконуються одночасно, для узгодження доступу до спільної пам’яті використовується бар’єрна синхронізація через функцію \_\_syncthreads(). Вона гарантує, що всі потоки блоку досягли певної точки виконання перед переходом далі, що критично важливо для коректного обміну даними.



Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.АО

малюнок 2.3 – Розширення мови С++

### **Етапи виконання CUDA-програми**

Типова програма з використанням CUDA виконується за наступним сценарієм:

1. Виділення пам’яті на пристрої (GPU).
2. Копіювання вхідних даних з пам’яті хоста в пам’ять пристрою.
3. Запуск обчислювального ядра (kernel) з визначеними параметрами блоків та сітки.
4. Копіювання результатів назад до хоста.
5. Звільнення пам’яті GPU.

Кожне ядро визначається як функція з модифікатором \_\_global\_\_ і викликається з CPU через синтаксис <<<gridSize, blockSize>>>.

### **Основні засоби розробки**

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

CUDA-програми зазвичай мають розширення .cu і компілюються за допомогою компілятора nvcc. CUDA розширює мову C низкою елементів:

* Специфікатори функцій, які визначають місце виконання функцій (на GPU або CPU);
* Типи пам’яті: глобальна, спільна, локальна, постійна;
* Директиви запуску ядер: конфігурують кількість потоків і блоків;
* Вбудовані змінні: надають ідентифікатори потоків і блоків;
* Бібліотеки runtime, які полегшують працю з пам’яттю синхронізацією та ядрами.

Ці можливості дозволяють реалізовувати високопродуктивні програми для обробки великих масивів даних з використанням GPU, забезпечуючи масштабованість та ефективність.

**2.2.1 Специфікатори функцій й змінні**

У середовищі розробки CUDA передбачено використання спеціальних специфікаторів функцій, які визначають, де і як саме буде виконуватись певна функція — на центральному процесорі (CPU) чи графічному процесорі (GPU). Основні специфікатори функцій наведені в таблиці 2.1.

|  |  |  |
| --- | --- | --- |
| Специфікатор | Функція використовується | Функція викликатися із |
| \_\_device\_\_ | (GPU)-devise | (GPU)-devise |
| \_\_global\_\_ | (GPU)-devise | (CPU)-host |
| \_\_ host\_\_ | (СPU)-host | (CPU)-host |

Таблиця 2.1 – Таблиця специфікацій

Зокрема, специфікатори \_\_host\_\_ та \_\_device\_\_ можуть застосовуватись одночасно до однієї функції. Це означає, що компілятор згенерує окремі версії коду для виконання CPU та на GPU. Такий підхід забезпечує універсальність використання однієї функціональної логіки на різних типах обчислювальних пристроїв.

Натомість специфікатори \_\_global\_\_ і \_\_host\_\_ є несумісними між собою, оскільки \_\_global\_\_ використовується виключно для позначення ядерної функції (kernel), яка викликається з CPU і виконується на GPU. Функції з цим специфікатором повинні повертати лише значення типу void і не можуть бути викликані з інших пристроїв.

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

Функції, позначені як \_\_device\_\_ або \_\_global\_\_, які виконуються на GPU, мають низку функціональних обмежень:

* неможливо отримати їхню адресу (виняток — глобальні функції);
* не підтримується рекурсивне викликання функцій;
* не дозволено використання статичних змінних усередині функції;
* заборонено використовувати змінну кількість вхідних параметрів (varargs).

Для управління розміщенням змінних у пам’яті GPU передбачені наступні специфікатори: \_\_device\_\_, \_\_constant\_\_ та \_\_shared\_\_. Кожен із них має свою специфіку використання:

* ці специфікатори не можуть бути застосовані до полів структур (типів struct або union);
* змінні, оголошені як \_\_constant\_\_, можуть бути ініціалізовані лише з боку хосту (CPU) через спеціалізовані API-функції;
* \_\_shared\_\_ змінні, призначені для спільного використання між потоками одного блоку, не можуть бути ініціалізовані під час оголошення.

Використання зазначених специфікаторів є ключовим аспектом при розробці високопродуктивних програм на CUDA, адже саме вони визначають модель пам’яті та правила виклику функцій у гетерогенному середовищі CPU-GPU.

### **2.2.2 Спеціальні змінні в CUDA**

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

У мові програмування CUDA передбачено використання низки вбудованих змінних, що дозволяють реалізовувати паралельні обчислення з урахуванням ієрархії сітки та блоків. Ці змінні є частиною середовища виконання ядра на GPU та автоматично доступні всередині коду, позначеного атрибутом \_\_global\_\_ або \_\_device\_\_. До таких змінних належать:

* gridDim — змінна типу dim3, що містить інформацію про розміри сітки, тобто кількість блоків у кожному з трьох вимірів (X, Y, Z);
* blockDim — також типу dim3, визначає розмір кожного окремого блоку в межах сітки, тобто кількість потоків у блоці по кожній з осей;
* blockIdx — змінна типу uint3, яка вказує індекс поточного блоку в загальній сітці блоків;
* threadIdx — змінна типу uint3, яка задає індекс потоку (нитки) в межах блоку;

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

* warpSize — змінна типу int, що визначає розмір одного warp’а, тобто групи потоків, які виконуються одночасно в одному SIMD-процесорі (за замовчуванням 32 потоки).

### **2.2.3 Синтаксис виклику ядра CUDA**

Запуск функції ядра (kernel) на графічному процесорі у CUDA здійснюється за допомогою спеціального синтаксису, що включає так звану директиву конфігурації ядра. Виглядає вона наступним чином:

kernelName<<<Dg, Db, Ns, S>>>(args);

Тут:

* kernelName — ім’я функції, позначеної атрибутом \_\_global\_\_, яка виконується на GPU;
* Dg — об’єкт типу dim3, який задає розміри сітки блоків (gridDim), тобто кількість блоків по кожній координаті;
* Db — об’єкт типу dim3, що визначає розміри окремого блоку (blockDim), тобто кількість потоків по кожному з трьох напрямків;
* Ns (необов’язковий параметр) — значення типу size\_t, яке визначає обсяг додаткової динамічної пам’яті (в байтах), що буде виділено для кожного блоку у спільній пам’яті (shared memory). Якщо цей параметр не задано, його значенням вважається 0;
* S (необов’язковий параметр) — змінна типу cudaStream\_t, яка визначає потік виконання (stream) для запуску ядра. За замовчуванням використовується потік з індексом 0;
* args — список аргументів, що передаються у виклик функції kernelName.

### **2.2.4 Розширені математичні функції в CUDA**

CUDA забезпечує підтримку широкого спектра математичних функцій, аналогічних функціям стандартної бібліотеки мови програмування C++. Проте слід враховувати, що багато з цих функцій реалізовані для роботи з числами подвійної точності (типу double). На сучасних графічних процесорах операції з типом double виконуються помітно повільніше, ніж аналогічні обчислення з типом float. Тому для досягнення вищої продуктивності доцільно застосовувати функції, оптимізовані для чисел одинарної точності (float).

Наприклад, замість стандартної функції sin, яка оперує значеннями double, можна використати sinf, призначену спеціально для float. Такий підхід дозволяє зменшити обчислювальні витрати та підвищити ефективність, особливо при роботі з великою кількістю паралельних потоків на GPU.

Крім цього, бібліотека CUDA включає спеціалізований набір функцій із зниженою точністю, які забезпечують ще вищу швидкодію. Ці функції орієнтовані на ситуації, коли допустима незначна втрата точності в обмін на суттєве зростання швидкості виконання. Згідно з документацією NVIDIA (NVIDIA Developer, n.d.), саме такі функції, як sinf, дозволяють максимально ефективно використовувати обчислювальний потенціал графічного процесора завдяки оптимізації під архітектуру GPU.

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

### **2.3 Висновки до розділу**

Технологія CUDA розроблена компанією NVIDIA з метою ефективної реалізації масово-паралельних обчислень на графічних процесорах. Вона забезпечує зручні засоби програмування, використовуючи розширення мови C++, що робить її доступною для широкого кола розробників. Підтримка охоплює всі відеокарти NVIDIA, починаючи з серії GeForce 8, а також професійні рішення, такі як серія Tesla, спеціалізовані для інтенсивних обчислювальних задач.

Серед ключових переваг CUDA варто виділити:

* Простота розробки — можливість писати код на розширеному C++ без глибокого занурення в архітектуру GPU.
* Широкий набір інструментів — включає компілятор nvcc, профайлери, бібліотеки для роботи з лінійною алгеброю, статистикою тощо.
* Платформна незалежність — підтримуються основні операційні системи: Windows, Linux, macOS.
* Відкритість екосистеми — повний комплект SDK, приклади коду, документація доступні безкоштовно на офіційному сайті [NVIDIA Developer](https://urldefense.com/v3/__https:/developer.nvidia.com__;!!AB_04_y_3-SRqw!6b6s8v2_uuJhwEhQhT6WnH8NspPwW3TrcuzLXcqITDLhgjjlrm8BJPAspCNJhRyeJpkQNbLPBjzaqWpFtSdUS2zyqwzPNA$).

На момент написання цього звіту актуальною є версія CUDA 2.3, хоча новіші версії вже мають розширені можливості та покращену продуктивність. CUDA-платформа забезпечує підтримку всіх основних математичних операцій, які наявні у стандартній бібліотеці C++, а також надає доступ до спеціалізованих, високопродуктивних реалізацій для типу float.

CUDA дозволяє реалізовувати обчислення за допомогою великої кількості паралельних ниток (threads), де кожна нитка зазвичай обробляє окремий елемент даних. Це забезпечує значне підвищення продуктивності у порівнянні з традиційними підходами, де використовується один або кілька потоків на CPU.

Таким чином, використання CUDA є доцільним для широкого спектра завдань, що передбачають складні обчмслення,наприклад,у сфері машинного навчаннч, комп’ютерна графіка, обробка зображень, симуляції фізичних процесів тощо.

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО



Змн.

Арк.

№ документа

Підпис

Дата

Аркуш

КНУ.РБ.123.20.01.АО

Розробив

Жижич

Перевірив

Купін

Н.контроль

Кузнєцов

Затвердив

Купін

ПРОЕКТУВАННЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

Літера

Аркушів

ЗКІ-21

## 3. ПРОЕКТУВАННЯ ТА ТЕСТУВАННЯ ПРОГРАМНОГО ЗАБЕЗПЕЧЕННЯ

### **3.1. Обґрунтування вибору мови програмування та середовища розробки**

У контексті створення високопродуктивного програмного забезпечення для обробки великих обсягів числових даних, особлива увага приділялася вибору відповідної мови програмування та середовища розробки. З урахуванням функціональних вимог проєкту, а також необхідності реалізації паралельної обробки, було обрано мову C# у поєднанні з інтегрованим середовищем розробки.

Програмування C# (вимовляється як “Сі Шарп”) належить до групи об’єктно-орієнтованих мов з потужним інструментарієм, орієнтованим на розробку додатків різного рівня складності. C# — це мова з суворою статичною типізацією, що реалізує цілу низку сучасних програмних парадигм, серед яких поліморфізм, наслідування, делегати, події, узагальнення (generics), анонімні методи, ітератори, LINQ-запити, механізми обробки виключень, XML-документування та інші.

Ключові переваги мови C#:

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

* Повна інтеграція з платформою .NET, що дозволяє ефективно використовувати її функціональність — від бібліотек класів (FCL) до керованого виконання на віртуальній машині.
* Мова є повністю орієнтованою на об’єкти: навіть найпростіші типи даних, такі як int або bool, реалізовані як об’єктні обгортки.
* Висока безпека пам’яті: C# обмежує використання вказівників і пряму роботу з адресами, надаючи більш захищене середовище, особливо у порівнянні з C++.
* Знайомий синтаксис для розробників, які мають досвід з C/C++, що спрощує перехід до нової мови.
* C# має вбудовані засоби для організації паралельного виконання програм завдяки зручним у застосуванні бібліотекам, таким як System.Threading та Task Parallel Library.

У порівнянні з Java, C# також компілюється у проміжний байт-код, однак працює під керуванням CLR, що дозволяє оптимізувати виконання завдяки механізмам JIT-компіляції. Це сприяє підвищенню продуктивності та ефективності виконання програм [2].

Щодо вибору інтегрованого середовища розробки, було обрано, воно забезпечує максимальну сумісність з мовою C# та платформою .NET. Серед альтернативних рішень можна згадати MonoDevelop, однак функціональні можливості Visual Studio значно ширші.

* Функція IntelliSense забезпечує контекстну підтримку під час кодування, пропонуючи підказки й автодоповнення синтаксису.
* Вбудований зневадник з підтримкою налагодження як на рівні вихідного коду, ї байт-коду.
* Інструменти для створення графічних інтерфейсів, редактори баз даних, модулі для проєктування класів.
* Можливості рефакторингу, що дозволяють легко покращувати структуру коду без зміни його функціональності.

Одним із важливих чинників у виборі Visual Studio 2017 стала висока продуктивність при роботі з великими проєктами. Завдяки широкій екосистемі, інтеграції з Azure, підтримці розширень та гнучкому налагодженню, це середовище дозволяє зручно створювати як консольні, так і графічні або веб-додатки, з використанням єдиного набору технологій.

Зовнішній вигляд Visual Studio наведено на рисунку 3.1.

Арк.

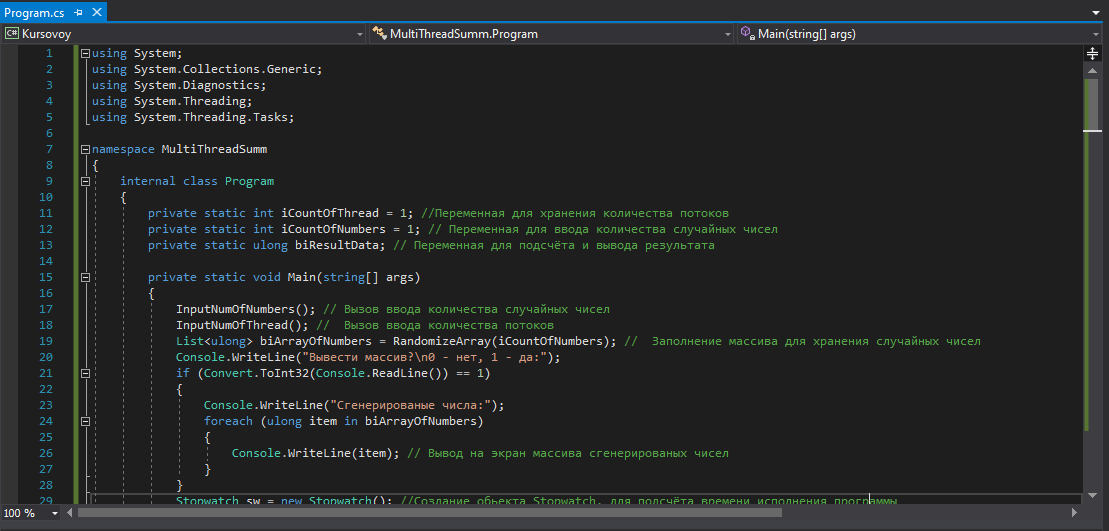
№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО



малюнок 3.1 – Зовнішній вигляд Visual Studio

### **3.2. Розробка програмного забезпечення**

На початковому етапі створення програмного забезпечення необхідно підключити необхідні простори імен, які забезпечують доступ до функціональних можливостей мови програмування C#. У контексті нашої програми використовуються п’ять основних просторів імен, кожен з яких виконує специфічні задачі. Хоча часто їх називають “бібліотеками”, з погляду синтаксису C# правильніше говорити саме про простори імен (namespaces), які містять відповідні класи та методи.

Підключення просторів Імен:

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Threading;

using System.Threading.Tasks;

* System — базовий простір імен, що містить фундаментальні класи, необхідні для більшості C#-програм (наприклад, Console, Math, DateTime тощо).
* System.Diagnostics — надає інструменти для діагностики та профілювання додатків. Зокрема, використовується клас Stopwatch для вимірювання часу виконання операцій.
* System.Threading використовується для управління потоками: запуску, зупинки та встановлення пріоритетів їх виконання.
* System.Threading.Tasks — надає API для асинхронного та паралельного програмування, зокрема клас Task, що дозволяє зручно працювати з асинхронними задачами.
* System.Collections.Generic — включає колекції з підтримкою типізації, такі як List<T>, Dictionary<TKey, TValue> тощо. Вони є ефективними та безпечними у порівнянні з неуніверсальними колекціями (ArrayList, Hashtable).

Примітка: Простори імен у C# подібні до бібліотек у Java або C++, однак у C# вони слугують для логічного групування типів.

#### **Оголошення глобальних змінни**

На наступному етапі оголошуються глобальні змінні, для управління потоками та збереження результатів обчислень:

Створемо глобальні змінні для роботи програми:

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

private static int iCountOfThread = 1;

private static int iCountOfNumbers = 1;

private static ulong biResultData

* iCountOfThread — визначає кількість потоків, які буде створено для паралельного виконання.
* iCountOfNumbers — які потрібно обробити (сумувати).
* biResultData — змінна типу ulong для збереження підсумкового результату обчислень.

#### Реалізація методу введення даних

Подальшим кроком є створення методу, який реалізує введення необхідних параметрів з клавіатури. Важливо передбачити перевірку введених значень на коректність (наприклад, щоб уникнути некоректного або негативного вводу), забезпечуючи тим самим стабільну роботу програми.

Основна точка входу:

private static void Main(string[] args)

{

InputNumOfNumbers(); // запуск обробки та вказання кількості випадкових значень

InputNumOfThread(); // ініціація роботи з визначеним числом потоків

List<ulong> biArrayOfNumbers = RandomizeArray(iCountOfNumbers); // створення масиву для збереження випадково згенерованих чисел

Console.WriteLine("Вивести масив?\n0 - ні, 1 - да:");

if (Convert.ToInt32(Console.ReadLine()) == 1)

{

Console.WriteLine("сгенеровані числа:");

foreach (ulong item in biArrayOfNumbers)

{

Console.WriteLine(item); // // вивести на єкран масив випадкових чисел

}

}

Stopwatch sw = new Stopwatch(); //Ініціалізація об’єкта Stopwatch, для вимірювання тривалості виконання програми

sw.Start(); // активація зазобу вимірювання часу

ulong[][] liBigInt = FillingListByCount(iCountOfThread, iCountOfNumbers, biArrayOfNumbers); // формування списків для паралельної обробки данних

//Console.WriteLine("counting array:"); //частина коду, яка показує розподіл даних по масивах,використовується для діагностики

//for (int i = 0; i < iCountOfThread; i++)

// for (int j = 0; j < liBigInt[0].Length; j++)

// Console.WriteLine(liBigInt[i][j]);

biResultData = MultiThreadCalculation(liBigInt); //Запуск багатопотокового процесу

sw.Stop();//запуск секундоміра

Console.WriteLine("Сума усіх чисел рівна: " + biResultData);

Console.WriteLine("Час пірахунку суми: " + sw.ElapsedMilliseconds / 1000 + " s " + sw.ElapsedMilliseconds % 1000 + " ms ");

Console.ReadKey();

}

Фрагмент коду для отримання кількості чисел:

private static void InputNumOfNumbers()//метод для введення кількості чисел

{

Console.WriteLine("введіть число випадкових значень для підрахунку:");

string input = Console.ReadLine();

try

{

int.TryParse(input, out iCountOfNumbers); // обробка введеня для отримання даних типа Integer

if (iCountOfNumbers < 1)//контроль правильності вводу інформації

{

Console.WriteLine("кількість чисел не може бути меньшою за одиницю!\nПрограма буде завершена!!!");

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.АО

Thread.Sleep(2000);//затримка перед завершенням

Environment.Exit(0);

}

}

catch (InvalidCastException e)

{

if (e.Source != null)

{

Console.WriteLine("помилка під час використання програми: {0}", e.Source); // помилки,що винекли в процесі конвертації

}

}

}

Код для ввення значення кількості потоків:

private static void InputNumOfThread()//спосіб задання кількості потоків

{

Console.WriteLine("отримання кількості потоків для функціонування програми:");

string input = Console.ReadLine();

try

{

int.TryParse(input, out iCountOfThread); // конвертація введених значень тип Integer

if (iCountOfThread <= 0)

{

Console.WriteLine("кількість потоків має бути не меньше за один!\виконання програми буде припенено!!!");

Thread.Sleep(2000); //Пауза перед завешенням роботи

Environment.Exit(0);

}

}

catch (InvalidCastException e)

{

if (e.Source != null)

{

Console.WriteLine("помилка у користуванні програмою: {0}", e.Source);

}

}

}

Код для створення масиву з випадковими числами:

private static List<ulong> RandomizeArray(int iCountInArray) // метод,який заповнює масив випадковими значеннями

{

List<ulong> biTempArray = new List<ulong>(); // тимчасовий масиввикористовується для збереження проміжних даних

Random FuncRand = new Random(); // створюється об’єкт Random який відповідає ініціалізацію масиву

for (int i = 0; i < iCountInArray; i++) // цикл призначений для заповнення кожного елементу масиву

{

biTempArray.Add(Convert.ToUInt64(FuncRand.Next(1000000))); // масив заповнюється випадковими числами

}

return biTempArray; // повертається готовий масив

}

Код для обчислення суми елементів списку та розподілу задачі на потоки:

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

private static ulong SummOfNumbers(ulong[] biInputData)// метод для обчислення суми елементів списку

{

ulong biResult = 0; // ініціалізація тимчасової змінної для збереження результату,який повертається з методу

foreach (ulong item in biInputData)

{

biResult += item; // підсімовування значень усіх елементів вхідного списку

}

return biResult;

}

private static ulong[][] FillingListByCount(int inNumOfThread, int inNumOfNumbers, List<ulong> biInputList)// метод для заповнення списку то розподілу завдання між кількома потоками

{

ulong[][] biOutResult = new ulong[inNumOfThread][];

if (inNumOfThread == 1) // запис даних в один масив,що відповідає одному потоку

{

biOutResult[0] = new ulong[inNumOfNumbers];

for (int i = 0; i < inNumOfNumbers; i++)

{

biOutResult[0][i] = biInputList[i];

}

return biOutResult;

}

else // якщо кількість потоків більша за один

{

for (int i = 0; i < inNumOfThread; i++)

{

biOutResult[i] = new ulong[inNumOfNumbers / inNumOfThread + 1]; // створюються окремімасиви для кожного потоку,відповідно до їхньої кількості

}

int iMinRang,

iMaxRang = inNumOfNumbers,

iMinusTemp,

iCounter;

if (inNumOfNumbers % inNumOfThread == 0)

{

iMinusTemp = inNumOfNumbers / inNumOfThread;

}

else

{

iMinusTemp = inNumOfNumbers / inNumOfThread + 1;

}

iMinRang = iMaxRang - iMinusTemp;

for (int i = inNumOfThread - 1; i >= 0; i--) // заповнюються відповідними даними

{

iCounter = 0;

for (int j = iMinRang; j < iMaxRang; j++)

{

biOutResult[i][iCounter] = biInputList[j];

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

iCounter++;

}

if (iMinRang - iMinusTemp > 0)

{

iMaxRang -= iMinusTemp;

iMinRang = iMaxRang - iMinusTemp;

}

else

{

iMinRang = 0;

iMaxRang -= iMinusTemp;

}

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

}

}

return biOutResult;

}

Ініціалізація потоків і обчислення результатів для кожного з них:

private static Task<ulong> AsyncTasks(ulong[] biArray) //Двикористання делегатів для ініціалізації потоків

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

{ return Task<ulong>.Run(() => SummOfNumbers(biArray)); }

private static ulong MultiThreadCalculation(ulong[][] packetForCalculating) //функція що реалізує паралельне обчислення у деяких потоках

{

Task<ulong>[] tasks = new Task<ulong>[iCountOfThread];

for (int i = 0; i < iCountOfThread; i++)

{

tasks[i] = AsyncTasks(packetForCalculating[i]); //активація асинхронних потоків через делегати

}

Task<ulong>.WaitAll(tasks); //очікуваннязавершення виконання усіх потоків

ulong result = 0;

for (int i = 0; i < iCountOfThread; i++)

{

result += tasks[i].Result; //об'єднання результатів,отриманих із кожного потоку

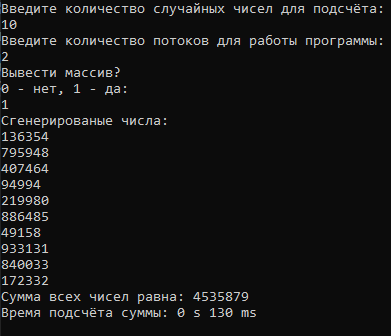
}

return result;

}

}

}



Малюнок 3.2 – Приклад виеонання програми

Запропонований метод не лише забезпечує розподіл обчислювальних задач між кількома потоками, але й реалізує динамічне формування блоків даних із варіативною довжиною інтервалів. Суть підходу полягає в тому, що зі збільшенням загального обсягу обчислюваних даних відбувається зменшення інтервалів кожного окремого пакету, що дозволяє рівномірніше завантажити обчислювальні потоки. Такий підхід сприяє ефективнішому використанню апаратних ресурсів системи та, відповідно, підвищенню загальної продуктивності програмного забезпечення.

Цей принцип динамічного балансування навантаження активно застосовується в сучасних обчислювальних системах. Наприклад, у рамках CUDA-архітектури NVIDIA реалізовано механізми динамічного завантаження GPU-кодів, що дозволяє адаптивно керувати обчислювальними ресурсами та оптимізувати виконання задач у реальному часі . Подібні підходи також використовуються в хмарних обчисленнях, де застосовуються алгоритми глибокого навчання з підкріпленням для оптимізації розподілу задач та зменшення енергоспоживання .

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

### **3.3 Оцінка рівня паралелізму програми**

Рівень ефективності паралельного виконання програми можна проаналізувати за допомогою закону Амдала. Формула для обчислення теоретичного прискорення виглядає наступним чином:

## 

де:

* S — коефіцієнт прискорення (speedup), що показує, у скільки разів прискорюється виконання програми завдяки паралельній обробці;
* P — частка програми, яка може бути паралелізована;
* N — кількість потоків або процесорів, що одночасно беруть участь в обчисленнях.

Цей закон описав Джин Амдал у 1967 році й він вказує на теоретичну межу прискорення, яке можна досягти при паралельній обробці завдань . Наприклад, якщо лише 50 % коду програми піддається розпаралеленню, то навіть при необмеженій кількості процесорів максимальне прискорення не перевищить дворазового.

Для наочності було побудовано графік залежності часу виконання програми від кількості потоків та обсягу вхідних даних (див.малюнок 3.3).

При незначних обсягах вхідних даних збільшення кількості потоків не завжди призводить до покращення продуктивності. Навпаки — через накладні витрати на керування потоками може спостерігатися навіть зниження ефективності. Це пояснюється тим, що ресурси витрачаються на організацію паралельного виконання, тоді як обчислювальне навантаження є занадто малим, аби ці витрати окупилися.

Проте в умовах обробки великих масивів даних ситуація змінюється: із зростанням обсягу задачі залучення більшої кількості потоків дає змогу суттєво скоротити загальний час виконання програми. Це обумовлено тим, що паралельні обчислення краще масштабуються при більших обсягах робочих даних, що, своєю чергою, підвищує загальну ефективність системи.

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

### **3.4 Висновки до розділу**

У розділі було представлено логіку роботи програмного забезпечення, спрямованого на реалізацію паралельних обчислень. Розроблено алгоритм функціонування програми, що дозволяє ефективно розподіляти обчислювальні завдання між кількома потокам.

Як основну мову програмування для реалізації розробки обрано C#, що забезпечує гнучкі можливості для роботи з багатопоточністю завдяки вбудованим засобам паралельного програмування (наприклад, простір імен System.Threading та Task Parallel Library). У процесі розробки було використано середовище Microsoft Visual Studio 2017, яке забезпечує ефективні інструменти для створення, відлагодження та тестування програм з підтримкою багатопотоковості.

Було створено програмний продукт, подано лістинг вихідного коду та проведено його аналіз з позицій ефективності реалізації паралельного підходу. У ході тестування визначено, що розпаралелювання обчислень дозволяє скоротити час виконання програми у порівнянні з послідовною реалізацією, особливо при обробці великих обсягів даних. Це підтверджує доцільність використання багатопотокової моделі в контексті задач високопродуктивних обчислень.

Окрім позитивних результатів, було також виявлено певні обмеження. Зокрема, при невдалому управлінні потоками можливе виникнення таких проблем, як стани гонки, блокування (deadlocks) або нерівномірне завантаження ресурсів, що може негативно впливати на стабільність роботи програми. Тому подальша оптимізація може полягати у впровадженні більш гнучких алгоритмів синхронізації та балансування навантаження.

Таким чином, проведене дослідження підтвердило ефективність використання багатопотокового підходу при створенні програм для паралельної обробки даних, а також окреслило напрями для подальшого вдосконалення програмного забезпечення.

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

## Висновки

Змн.

Арк.

№ документа

Підпис

Дата

Аркуш

КНУ.РБ.123.20.01.ВС

Розробив

Жижич

Перевірив

Купін

Н.контроль

Кузнєцов

Затвердив

Купін

ВИСНОВКИ

Літера

Аркушів

ЗКІ-21

У результаті виконання переддипломної практики було поглиблено теоретичні та практичні знання з використання архітектури Технологія CUDA (Compute Unified Device Architecture), розроблена компанією NVIDIA, слугувала основою для реалізації паралельних обчислень із використанням графічних процесорів, що дозволяє значно покращити продуктивність за рахунок одночасної обробки великої кількості даних.

Платформа підтримує низку мов програмування, серед яких C, C++, Python, Java тощо, що забезпечує її виконання програм у CUDA побудована на багаторівневій організації потоків: решітка (grid) складається з блоків (block), які в свою чергу містять окремі потоки (thread). Такий підхід дозволяє паралельно обробляти великі обсяги інформації.

Технологія широко застосовується в сучасних сферах, зокрема в штучному інтелекті, машинному навчанні, медичній візуалізації, тривимірному моделюванні, біоінформатиці, фінансовому аналізі тощо. Як свідчать приклади на офіційному сайті розробника, архітектура CUDA є основою як експериментальних, так і комерційних програмних рішень .

## Розробка алгоритму та програмної реалізації

У другому розділі роботи було розроблено алгоритм, що виконує паралельне обчислення суми з N довільних чисел. Основною мовою програмування для реалізації проєкту було обрано C#, а середовищем розробки — Microsoft Visual Studio 2017, яке забезпечило всі необхідні засоби для побудови ефективного багатопотокового застосунку.

Розроблений програмний продукт забезпечує:

* генерацію випадкових чисел;
* розподіл даних між потоками;
* паралельне виконання обчислень;
* об’єднання часткових результатів у підсумкове значення.

Лістинг коду наведено у відповідному розділі та супроводжено поясненнями щодо його роботи.

## Тестування та аналіз ефективності

Тестування реалізованої програми проводилось із використанням різних обсягів вхідних даних. Було встановлено, що за рахунок багатопотокового виконання вдається значно скоротити час обробки, що підтверджує ефективність застосування підходу до паралелізації.

Програма показала стійкість до змін вхідних параметрів та правильність отриманих результатів. Водночас були проаналізовані технічні обмеження: зменшення ефективності при перевищенні оптимальної кількості потоків, навантаження на ресурси системи, потреба у синхронізації обчислень.

Загалом отримані результати засвідчили доцільність використання паралельних обчислень на основі архітектури CUDA або схожих підходів при розв’язанні задач, що передбачають великі обсяги однотипних операцій.

Змн.

Арк.

№ документа

Підпис

Дата

Аркуш

КНУ.ПК.123.24.18.АО

Розробив

*Жижич*

Купін

Перевірив

Н.контроль

Кузнєцов

Затвердив

Купін

ВИСНОВКИ

Літера

Аркушів

ЗКІ-21

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

Змн.

Арк.

№ документа

Підпис

Дата

Аркуш

КНУ.РБ.123.20.01.ВС

Розробив

Жижич

Перевірив

Купін

Н.контроль

Кузнєцов

Затвердив

Купін

СВД

Літера

Аркушів

КІ-16-1

1. NVIDIA. Документація до інструментарію CUDA [Електронний ресурс]. – Режим доступу: [https://developer.nvidia.com/cuda-toolkit](https://urldefense.com/v3/__https:/developer.nvidia.com/cuda-toolkit__;!!AB_04_y_3-SRqw!8XhesZx_Fj4x5lr4Czh20fmExQDT0hHwySvnT4uWWft_0OST0dXZz5q-oZZkJOpcBLkNr9I5sFFDxoXwbBens7aO0PsHOQ$) (дата звернення: 04.04.2025).
2. Сандерс Д., Кандрот Е. CUDA на прикладах: вступ до універсального програмування для графічних процесорів. – Вид. Addison-Wesley, 2010. – 312 с.
3. Кірк Д.Б., Хву В.В. Програмування масово паралельних процесорів: практичний підхід. – 3-тє вид. – Видавництво Morgan Kaufmann, 2016. – 576 с.
4. Microsoft. Основи багатопотокового програмування в C# [Електронний ресурс]. – Режим доступу: <https://learn.microsoft.com/en-us/dotnet/standard/threading/> (дата звернення: 04.04.2025).
5. Глушаков С.О. Паралельні обчислення із застосуванням графічних процесорів у середовищі C#. – Харків: ХНУРЕ, 2020. – 144 с.
6. Intel. Технологія Intel® Hyper-Threading: принципи роботи та призначення [Електронний ресурс]. – Режим доступу: [https://www.intel.ru/content/www/ru/ru/architecture-and-technology/hyper-threading/hyper-threading-technology.html](https://urldefense.com/v3/__https:/www.intel.ru/content/www/ru/ru/architecture-and-technology/hyper-threading/hyper-threading-technology.html__;!!AB_04_y_3-SRqw!8XhesZx_Fj4x5lr4Czh20fmExQDT0hHwySvnT4uWWft_0OST0dXZz5q-oZZkJOpcBLkNr9I5sFFDxoXwbBens7ZUen3dDQ$) (дата звернення: 04.04.2025).
7. OpenMP. Офіційний вебсайт проекту [Електронний ресурс]. – Режим доступу: [https://www.openmp.org/](https://urldefense.com/v3/__https:/www.openmp.org/__;!!AB_04_y_3-SRqw!8XhesZx_Fj4x5lr4Czh20fmExQDT0hHwySvnT4uWWft_0OST0dXZz5q-oZZkJOpcBLkNr9I5sFFDxoXwbBens7YebMl1xg$) (дата звернення: 04.04.2025).
8. The Perils of Parallel. Порівняння Larrabee та NVIDIA: архітектура MIMD проти SIMD [Електронний ресурс]. – Режим доступу: [http://perilsofparallel.blogspot.com/2008/09/larrabee-vs-nvidia-mimd-vs-simd.html](https://urldefense.com/v3/__http:/perilsofparallel.blogspot.com/2008/09/larrabee-vs-nvidia-mimd-vs-simd.html__;!!AB_04_y_3-SRqw!8XhesZx_Fj4x5lr4Czh20fmExQDT0hHwySvnT4uWWft_0OST0dXZz5q-oZZkJOpcBLkNr9I5sFFDxoXwbBens7a4qF5l0g$) (дата звернення: 04.04.2025).
9. Intel Developer Zone. Платформа для сучасного кодування [Електронний ресурс]. – Режим доступу: [https://software.intel.com/en-us/modern-code](https://urldefense.com/v3/__https:/software.intel.com/en-us/modern-code__;!!AB_04_y_3-SRqw!8XhesZx_Fj4x5lr4Czh20fmExQDT0hHwySvnT4uWWft_0OST0dXZz5q-oZZkJOpcBLkNr9I5sFFDxoXwbBens7bNXR0nxw$) (дата звернення: 04.04.2025).
10. Михайлов Б.М., Халабія Р.Ф. Класифікація та побудова обчислювальних систем: навчальний посібник. – Москва: МГУПІ, 2010. – 144 с. – [Національна бібліотека ім. Н. Е. Баумана] [Електронний ресурс]. – Режим доступу: [https://ru.bmstu.wiki/](https://urldefense.com/v3/__https:/ru.bmstu.wiki/__;!!AB_04_y_3-SRqw!8XhesZx_Fj4x5lr4Czh20fmExQDT0hHwySvnT4uWWft_0OST0dXZz5q-oZZkJOpcBLkNr9I5sFFDxoXwbBens7ZhDXQelw$) (дата звернення: 04.04.2025).
11. Habr. Основи багатопоточності в мові C# [Електронний ресурс]. – Режим доступу: [https://habr.com/ru/post/126495/](https://urldefense.com/v3/__https:/habr.com/ru/post/126495/__;!!AB_04_y_3-SRqw!8XhesZx_Fj4x5lr4Czh20fmExQDT0hHwySvnT4uWWft_0OST0dXZz5q-oZZkJOpcBLkNr9I5sFFDxoXwbBens7Yi0pdm2A$) (дата звернення: 04.04.2025).

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.РБ.123.20.01.ВC

1. Microsoft. Документація по структурі BigInteger у .NET [Електронний ресурс]. – Режим доступу: <https://docs.microsoft.com/ru-ru/dotnet/api/system.numerics.biginteger> (дата звернення: 04.04.2025).
2. Microsoft. Документація щодо роботи з Task у C# [Електронний ресурс]. – Режим доступу: <https://docs.microsoft.com/ru-ru/dotnet/api/system.threading.tasks.task> (дата звернення: 04.04.2025).
3. Intel. Основи програмування з використанням багатопотоковості: огляд підходів для різних мов [Електронний ресурс]. – Режим доступу: [https://software.intel.com/ru-ru/articles/writing-parallel-programs-a-multi-language-tutorial-introduction](https://urldefense.com/v3/__https:/software.intel.com/ru-ru/articles/writing-parallel-programs-a-multi-language-tutorial-introduction__;!!AB_04_y_3-SRqw!8XhesZx_Fj4x5lr4Czh20fmExQDT0hHwySvnT4uWWft_0OST0dXZz5q-oZZkJOpcBLkNr9I5sFFDxoXwbBens7YqkuWFYw$) (дата звернення: 04.04.2025).

ДОДАТОК А

using System;

using System.Collections.Generic;

using System.Diagnostics;

using System.Threading;

using System.Threading.Tasks;

namespace MultiThreadSumm

{

internal class Program

{

private static int iCountOfThread = 1; //кількість потоків

private static int iCountOfNumbers = 1; // кількість випадкових чисел

private static ulong biResultData; // зміна для збереження суми

private static void Main(string[] args)

{

InputNumOfNumbers(); // запуск процедури введеня кількості випадкових чисел

InputNumOfThread(); // ініціаліізація параметра кількості потоків

List<ulong> biArrayOfNumbers = RandomizeArray(iCountOfNumbers); // формування масиву для збереження випадкових значень

Console.WriteLine("відобразита згенерований список на екрані?\n0 - ні, 1 - так:");

if (Convert.ToInt32(Console.ReadLine()) == 1)

Змн.

Арк.

№ документа

Підпис

Дата

Аркуш

КНУ.ПК.123.24.18.АО

Розробив

*Жижич*

Купін

Перевірив

Н.контроль

Кузнецов

Затвердив

Купін

ДОДАТОК А

Літера

Аркушів

ЗКІ-21

{

Console.WriteLine("згенеровані числа:");

foreach (ulong item in biArrayOfNumbers)

{

Console.WriteLine(item); // відображення на екрані згенерованого набору чисел

}

}

Stopwatch sw = new Stopwatch(); //ініціалізація інструменту Stopwatch, для вимірювання тривалості виконання програми

sw.Start(); // пачаток відліку часу виконання

ulong[][] liBigInt = FillingListByCount(iCountOfThread, iCountOfNumbers, biArrayOfNumbers); // формування масивів данних для обробки у багатопоточному режимі

//Console.WriteLine("counting array:"); //фрагмент коду,що дозволяє перевірити розподіл даних між масивами для налагодження

//for (int i = 0; i < iCountOfThread; i++)

// for (int j = 0; j < liBigInt[0].Length; j++)

// Console.WriteLine(liBigInt[i][j]);

biResultData = MultiThreadCalculation(liBigInt); //старт паралельного обчислення за допомогою кількості потоків

sw.Stop();//зупинка вимірювання часу виконання

Console.WriteLine("результат обчислення сум всіх чисел становить: " + biResultData);

Console.WriteLine("загальний час, витрачений на підрахунок суми: " + sw.ElapsedMilliseconds / 1000 + " s " + sw.ElapsedMilliseconds % 1000 + " ms ");

Console.ReadKey();

}

private static void InputNumOfNumbers()//реалізація функції для задання розміру масиву чисел

{

Console.WriteLine("укажіть,скільки випадкових чисел потрібно згенурувати для обчислення суми:");

string input = Console.ReadLine();

try

{

int.TryParse(input, out iCountOfNumbers); // конвертація введенного значення у числовий формат Integer

if (iCountOfNumbers < 1)//контроль правильності введеня інформації

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

{

Console.WriteLine("кількість потоків має бути не меньше за один!\виконання програми буде припенено!!!");

Thread.Sleep(2000);//очікування перед закриттям вікна застосунку

Environment.Exit(0);

}

}

catch (InvalidCastException e)

{

if (e.Source != null)

{

Console.WriteLine("виявленно неправильне використання функціоналу програми: {0}", e.Source); // виникла помилка під час спроби перетворення введеної інформації

}

}

}

private static void InputNumOfThread()//функція,яка дозволяєзадати число потоків для паралельної обробки

{

Console.WriteLine("укажіть,скільки потоків меє бути використано при обчисленнях:");

string input = Console.ReadLine();

try

{

int.TryParse(input, out iCountOfThread); // перевелдення рядка у ціле число типа Integer

if (iCountOfThread <= 0)

{

Console.WriteLine("кількість потоків має бути не меньше за один!\виконання програми буде припенено!!!");

Thread.Sleep(2000); //пауза перед завершенням виконання

Environment.Exit(0);

}

}

catch (InvalidCastException e)

{

if (e.Source != null)

{

Console.WriteLine("неправильна робота з додатком призвела до помілки: {0}", e.Source);

}

}

}

private static List<ulong> RandomizeArray(int iCountInArray) // функція, що генерує масив із випадкових значень

{

List<ulong> biTempArray = new List<ulong>(); // масив,створенний для тимчасового збереження згенерованих даних

Random FuncRand = new Random(); // ініціалізація об’єкта Random для генерації чмсел

for (int i = 0; i < iCountInArray; i++) // проходження по масиву з метою його наповнення

{

biTempArray.Add(Convert.ToUInt64(FuncRand.Next(1000000))); // генераціявипадкових чисел і занесення їх у масив

}

return biTempArray; // метод формує масив як вихідне значення

}

private static ulong SummOfNumbers(ulong[] biInputData)// функція підрахунку загального значення з колекції

{

ulong biResult = 0; // результат функції заноситься до локальної змінної

foreach (ulong item in biInputData)

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

{

biResult += item; // агрегування значень,що находять у колекції

}

return biResult;

}

private static ulong[][] FillingListByCount(int inNumOfThread, int inNumOfNumbers, List<ulong> biInputList)// здіснюється заповнення масиву і паралельний розподіл навантаження

{

ulong[][] biOutResult = new ulong[inNumOfThread][];

if (inNumOfThread == 1) // кожному потоку призначається власна частина даних у масиві

{

biOutResult[0] = new ulong[inNumOfNumbers];

for (int i = 0; i < inNumOfNumbers; i++)

{

biOutResult[0][i] = biInputList[i];

}

return biOutResult;

}

else // якщо потоків більше одного

{

for (int i = 0; i < inNumOfThread; i++)

{

biOutResult[i] = new ulong[inNumOfNumbers / inNumOfThread + 1]; // кількість масивів відповідає числу потоків

}

int iMinRang,

iMaxRang = inNumOfNumbers,

iMinusTemp,

iCounter;

if (inNumOfNumbers % inNumOfThread == 0)

{

iMinusTemp = inNumOfNumbers / inNumOfThread;

}

else

{

iMinusTemp = inNumOfNumbers / inNumOfThread + 1;

}

iMinRang = iMaxRang - iMinusTemp;

for (int i = inNumOfThread - 1; i >= 0; i--) // формування масиву із заданими даними

{

iCounter = 0;

for (int j = iMinRang; j < iMaxRang; j++)

{

biOutResult[i][iCounter] = biInputList[j];

iCounter++;

}

if (iMinRang - iMinusTemp > 0)

{

iMaxRang -= iMinusTemp;

iMinRang = iMaxRang - iMinusTemp;

}

else

{

iMinRang = 0;

iMaxRang -= iMinusTemp;

}

}

}

return biOutResult;

}

private static Task<ulong> AsyncTasks(ulong[] biArray) //делегати ініціюють виконання окремих потоків

{ return Task<ulong>.Run(() => SummOfNumbers(biArray)); }

Арк.

№ документа

Підпис

Дата

Арк.

КНУ.ПК.123.24.18.АО

private static ulong MultiThreadCalculation(ulong[][] packetForCalculating) //метод обробки даних з розпаралелюванням

{

Task<ulong>[] tasks = new Task<ulong>[iCountOfThread];

for (int i = 0; i < iCountOfThread; i++)

{

tasks[i] = AsyncTasks(packetForCalculating[i]); //ініціалізація делегатів для виконання у фоновому режимі

}

Task<ulong>.WaitAll(tasks); //чекаємо,доки не буде завершено обчислення в кожному потоці

ulong result = 0;

for (int i = 0; i < iCountOfThread; i++)

{

result += tasks[i].Result; //фінальне об'єднання результатів з паралельних прцесів

}

return result;

}

}

}