ДЕРЖАВНИЙ УНІВЕРСИТЕТ
ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ
НАВЧАЛЬНО-НАУКОВИЙ ІНСТИТУТ
ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА КОМП'ЮТЕРНОЇ ІНЖЕНЕРІЇ

КВАЛІФІКАЦІЙНА РОБОТА

на тему: «Багатоядерні процесори у сучасних ігрових додатках»

на здобуття освітнього ступеня бакалавра
зі спеціальності 123 Комп'ютерна інженерія
освітньо-професійної програми «Комп'ютерна інженерія»

Кваліфікаційна робота містить результати власних досліджень. Використання ідей, результатів і текстів інших авторів мають посилання на відповідне джерело

_____ Олександр БАНДУРА
(підпис)

Виконав: здобувач вищої освіти групи КІД-41
Олександр БАНДУРА

Керівник: Сергій КОРОТКОВ

Рецензент: _____
Ім'я, ПРІЗВИЩЕ

Київ 2025

ДЕРЖАВНИЙ УНІВЕРСИТЕТ

ІНФОРМАЦІЙНО-КОМУНІКАЦІЙНИХ ТЕХНОЛОГІЙ

Навчально-науковий інститут інформаційних технологій

Кафедра Комп'ютерна інженерія

Ступінь вищої освіти Бакалавра

Спеціальність 123 Комп'ютерна інженерія

Освітньо-професійна програма «Комп'ютерна інженерія»

ЗАТВЕРДЖУЮ
Завідувач кафедри
Комп'ютерної інженерії
_____Наталія ЛАЩЕВСЬКА
«_____» _____2025 р.

ЗАВДАННЯ
НА КВАЛІФІКАЦІЙНУ РОБОТУ

Бандура Олександр Володимирович

1. Тема кваліфікаційної роботи: «Багатоядерні процесори у сучасних ігрових додатках»
керівник кваліфікаційної роботи Сергій КОРОТКОВ,

затверджені наказом Державного університету інформаційно-комунікаційних технологій від «24» лютого 2025 р. № 56.
2. Строк подання кваліфікаційної роботи «05» червня 2025 р.
3. Вихідні дані до кваліфікаційної роботи: науково-технічна література, технічна документація на багатоядерні процесори, стандарти розробки ігрових додатків, моделі паралельного програмування, методики оптимізації продуктивності, дані тестування продуктивності CPU, засоби реалізації багатопотокових обчислень.

4. Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)
1. Огляд літератури та аналіз предметної області у сфері використання багатоядерних процесорів в ігрових додатках.
2. Аналіз архітектурних особливостей багатоядерних процесорів та моделей паралельного програмування, що використовуються в ігрових застосунках.
3. Аналіз типових рішень щодо інтеграції багатоядерності в ігрову логіку та обробку ресурсомістких задач.

4. Оцінювання ефективності застосування багатоядерних процесорів у реалізації ключових ігрових сценаріїв на прикладі існуючих прикладних реалізацій.

1. Перелік ілюстративного матеріалу: *презентація*
Проблематика обмеженої продуктивності в ігрових додатках та потреба у багатоядерній обробці.
Огляд архітектури багатоядерних процесорів та принципів паралельної обробки даних.
Архітектура типових ігрових рушіїв із підтримкою багатопотоковості.
Приклади застосування багатоядерності в обробці ігрової логіки та фізики.
Результати тестування продуктивності: графіки навантаження на ядра CPU.
Порівняння ефективності з традиційними однопоточними реалізаціями: висновки та рекомендації.

Дата видачі завдання «24» лютого 2025 р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Строк виконання етапів роботи	Примітка
1	Аналіз науково-технічної літератури з питань використання багатоядерних процесорів в ігрових додатках	16.02–23.02.2025	
2	Вивчення сучасних архітектур багатоядерних процесорів та моделей паралельного програмування	23.02–02.03.2025	
3	Аналіз підходів до реалізації багатопотокових обчислень в ігрових середовищах	02.03–09.03.2025	
4	Оцінювання ефективності існуючих рішень щодо розподілу навантаження у багатоядерних системах	09.03–19.03.2025	
5	Аналіз продуктивності програмних рішень на базі CPU з різною кількістю ядер	19.03–29.03.2025	
6	Оформлення роботи: вступ, висновки, реферат	29.03–30.03.2025	
7	Підготовка демонстраційних матеріалів та презентації	30.03–31.03.2025	
8	Попередній захист роботи	01.04.2025	

Здобувач вищої освіти _____ Олександр БАНДУРА
(підпис)

Керівник
кваліфікаційної роботи _____
Сергій КОРОТКОВ
(підпис)

РЕФЕРАТ

Текстова частина бакалаврської роботи: 56 сторінок, 14 рисунків, 45 джерел.

Об'єкт дослідження – процеси обробки та оптимізації обчислювальних задач в ігрових додатках із використанням багатоядерних процесорів.

Предмет дослідження – архітектура багатоядерних процесорів, моделі паралельного програмування та програмні засоби реалізації багатопотокових обчислень у сучасних ігрових середовищах.

Мета роботи – проаналізувати можливості багатоядерних процесорів у підвищенні продуктивності ігрових застосунків, розробити та протестувати програмний модуль із підтримкою паралельної обробки даних, а також провести оцінку ефективності реалізованих рішень.

Методи дослідження:

1. Аналіз архітектурних особливостей сучасних CPU та моделей паралельного програмування.
2. Вивчення засобів розробки багатопотокових застосунків (OpenMP, Java Threads, ForkJoinPool).
3. Розробка демонстраційного програмного модуля з використанням паралельної логіки.
4. Проведення експериментального тестування з метою порівняння продуктивності на різних конфігураціях CPU.

Галузь використання – розробка високопродуктивних ігрових застосунків, ігрових рушіїв, симуляцій та програмних рішень, що потребують ефективного використання апаратних ресурсів багатоядерних процесорів.

КЛЮЧОВІ СЛОВА: БАГАТОЯДЕРНИЙ ПРОЦЕСОР, ПАРАЛЕЛЬНІ ОБЧИСЛЕННЯ, ІГРОВИЙ ДОДАТОК, МУЛЬТИПОТОКОВІСТЬ, ОПТИМІЗАЦІЯ КОДУ, JAVA THREADS, OPENMP, ВИСОКА ПРОДУКТИВНІСТЬ, CPU, ПАРАЛЕЛІЗМ В ІГРАХ

ABSTRACT

The textual part of the bachelor's thesis: 56 pages, 14 figures, 45 references.

Object of the research – processes of processing and optimization of computational tasks in gaming applications using multi-core processors.

Subject of the research – architecture of multi-core processors, models of parallel programming, and software tools for implementing multithreaded computations in modern gaming environments.

Purpose of the thesis – to analyze the capabilities of multi-core processors in enhancing the performance of gaming applications, to develop and test a software module supporting parallel data processing, and to evaluate the effectiveness of the implemented solutions.

Research methods:

1. Analysis of architectural features of modern CPUs and models of parallel programming.
2. Study of tools for multithreaded application development (OpenMP, Java Threads, ForkJoinPool).
3. Development of a demonstration software module using parallel logic.
4. Experimental testing to compare performance across different CPU configurations.

Field of application – development of high-performance gaming applications, game engines, simulations, and software solutions that require efficient use of multi-core processor hardware resources.

KEYWORDS: MULTI-CORE PROCESSOR, PARALLEL COMPUTING, GAME APPLICATION, MULTITHREADING, CODE OPTIMIZATION, JAVA THREADS, OPENMP, HIGH PERFORMANCE, CPU, GAME PARALLELISM

ЗМІСТ

ВСТУП 9

1 ТЕОРЕТИЧНІ ОСНОВИ ВИКОРИСТАННЯ БАГАТОЯДЕРНИХ ПРОЦЕСОРІВ У ПАРАЛЕЛЬНИХ ОБЧИСЛЕННЯХ 11

1.1 Основи паралелізму в обчислювальних системах 11

1.2 Архітектура багатоядерних процесорів: CPU vs GPU 15

1.3 Підходи до програмування багатоядерних систем на Java 22

1.4 Огляд технологій Java для високопродуктивних обчислень (Aparapi, TornadoVM) 24

2 ПРОЄКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО МОДУЛЯ ДЛЯ ОБРОБКИ ІГРОВИХ ЗАДАЧ НА CPU 28

2.1 Вибір інструментів і бібліотек для реалізації багатопотоковості 28

2.2 Архітектура і структура розробленої системи 30

2.3 Інтеграція та взаємодія потоків у задачах обробки ігрової логіки 33

2.4 Оптимізація навантаження на ядра процесора 34

3 ТЕСТУВАННЯ ЕФЕКТИВНОСТІ ТА ОЦІНКА ПРОДУКТИВНОСТІ ЗАПРОПОНОВАНОГО РІШЕННЯ 37

3.1 Методика тестування: критерії, середовище, сценарії 37

3.2 Порівняльний аналіз продуктивності (багатоядерність vs однопоточність) 41

3.3 Аналіз стабільності, масштабованості та виявлених обмежень 48

3.4 Перспективи розвитку і застосування ігрових CPU-систем 53

ВИСНОВКИ 55

ПЕРЕЛІК ПОСИЛАНЬ 56

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ 60

ВСТУП

Сучасні ігрові додатки потребують високої продуктивності для обробки складної графіки, фізичних моделей та логіки взаємодії. Одним із ключових шляхів досягнення цього є ефективне використання багатоядерних процесорів. Хоча більшість сучасних CPU підтримують багатопотокову обробку, значна частина ігрових рішень не повною мірою реалізує її потенціал. Це зумовлює потребу в дослідженні архітектур багатоядерних процесорів, моделей паралельного програмування та методів оптимізації продуктивності. Додатковим фактором, що підсилює значущість дослідження, є потреба у зниженні затримок, збільшенні частоти кадрів (FPS), покращенні обробки ШІ та фізичних моделей у режимі реального часу. Саме багатоядерність і правильно організований паралелізм можуть дати змогу вирішити ці завдання без істотного збільшення енергоспоживання або необхідності в дорогому апаратному оновленні.

Розробка ігор, які здатні повністю використовувати переваги багатоядерності, є актуальним завданням як для індустрії, так і для прикладної науки.

Мета роботи – проаналізувати можливості багатоядерних процесорів у підвищенні продуктивності ігрових застосунків, розробити та протестувати програмний модуль із підтримкою паралельної обробки даних, а також провести оцінку ефективності реалізованих рішень.

Об’єкт дослідження – процеси обробки та оптимізації обчислювальних задач в ігрових додатках із використанням багатоядерних процесорів.

Предмет дослідження – архітектура багатоядерних процесорів, моделі паралельного програмування та програмні засоби реалізації багатопотокових обчислень у сучасних ігрових середовищах.

Методологія дослідження ґрунтується на аналізі та узагальненні науково-технічної літератури з питань архітектури багатоядерних процесорів, моделей паралельного програмування та принципів оптимізації продуктивності в ігрових системах. Особливу увагу приділено дослідженню засобів реалізації

багатопотокових обчислень, таких як Java Threads, ForkJoinPool, OpenMP, а також порівнянню їх ефективності в контексті ігрових додатків.

У межах дослідження було проведено порівняльний аналіз програмних моделей з урахуванням різних конфігурацій CPU, а також розглянуто способи організації паралелізму для обробки графіки, фізичних об’єктів і внутрішньоігрових подій. Застосовано методи системного аналізу, програмного моделювання, об’єктно-орієнтованого підходу до розробки та комп’ютерного експерименту.

Практична значущість роботи полягає в обґрунтуванні підходів до ефективного використання багатоядерних процесорів при розробці сучасних ігрових додатків. Проведене дослідження дозволяє розробникам краще розуміти вплив архітектури CPU на продуктивність програмних рішень та адаптувати свої проєкти для повноцінного використання багатопоточності.

Результати дослідження можуть бути використані для оптимізації існуючих ігрових рушіїв, проєктування нових програмних компонентів з підтримкою паралельної обробки, а також для навчання майбутніх фахівців у сфері гейм-розробки. Запропоновані висновки сприяють підвищенню якості користувацького досвіду, зниженню затримок і ефективнішому використанню апаратних ресурсів без необхідності в дорогому технічному оновленні.

1 ТЕОРЕТИЧНІ ОСНОВИ ВИКОРИСТАННЯ БАГАТОЯДЕРНИХ ПРОЦЕСОРІВ У ПАРАЛЕЛЬНИХ ОБЧИСЛЕННЯХ

Сучасна ігрова індустрія висуває дедалі вищі вимоги до апаратного забезпечення, що стимулює активне використання багатоядерних процесорів. Для повноцінного розкриття їхнього потенціалу необхідно глибоко розуміти принципи паралельної обробки даних, архітектурні особливості процесорів та відповідні програмні підходи.

1.1 Основи паралелізму в обчислювальних системах

Паралелізм – це здатність комп’ютерної системи одночасно виконувати кілька операцій або обчислювальних процесів. Він реалізується шляхом розподілу задач між декількома виконавчими одиницями, такими як ядра процесора чи обчислювальні блоки графічного процесора. Основна мета паралелізму – підвищення продуктивності, скорочення часу обробки та ефективніше використання ресурсів [1].

У сучасних комп’ютерних системах паралелізм є критично важливим через:

- зростання обсягів даних і складності задач, які неможливо ефективно опрацювати послідовно;
- фізичні обмеження частоти процесорів, які більше не можуть істотно збільшуватися;
- наявність багатоядерних архітектур, які надають апаратну основу для реалізації паралельних обчислень;
- вимоги до реального часу, наприклад, в ігровій графіці, симуляціях фізики чи штучному інтелекті, де затримки неприпустимі.

Паралелізм в обчислювальних системах є фундаментальним принципом, який дозволяє підвищити ефективність обробки даних за рахунок одночасного виконання декількох операцій або задач. Цей підхід став особливо актуальним із

розвитком багатоядерних процесорів, які надають фізичну можливість розподілу обчислювального навантаження між ядрами. У загальному випадку паралелізм поділяється на кілька рівнів реалізації, зокрема на рівні інструкцій, потоків і процесів, кожен з яких має свої особливості, переваги та області застосування.

Паралелізм на рівні інструкцій (Instruction-Level Parallelism, ILP) передбачає виконання декількох машинних інструкцій одночасно в межах одного потоку керування [2]. Цей тип паралелізму реалізується переважно апаратними засобами центрального процесора за допомогою таких технологій, як конвеєризація (pipelining), суперскалярна архітектура та позачергове виконання інструкцій (out-of-order execution). Наприклад, у суперскалярних процесорах декілька інструкцій можуть бути видані на виконання в один і той самий тактовий цикл, що дозволяє значно збільшити продуктивність без залучення додаткових ядер або потоків. Однак ефективність ILP суттєво залежить від структури коду, наявності залежностей між інструкціями та ефективності модулів прогнозування переходів.

Паралелізм на рівні потоків (Thread-Level Parallelism, TLP) базується на ідеї одночасного виконання декількох потоків в межах одного процесу. Потік (thread) – це найлегша одиниця виконання, яка має власний стек та регістри, але спільно використовує пам’ять із іншими потоками того самого процесу. Цей вид паралелізму є ключовим для сучасних багатоядерних процесорів, які дозволяють фізично розміщувати різні потоки на окремих ядрах [3]. Програмні засоби для реалізації TLP включають механізми багатопотокового програмування, такі як Java Threads, C++ std::thread, OpenMP, POSIX threads тощо. Потоки широко використовуються в ігрових додатках для розподілу задач між системами: наприклад, фізика, графіка, мережеві запити та логіка гравця можуть оброблятися паралельно.

Паралелізм на рівні процесів (Process-Level Parallelism, PLP) передбачає одночасне виконання декількох незалежних процесів, кожен з яких має власний адресний простір та набір системних ресурсів. Цей підхід застосовується у випадках, коли задачі не залежать одна від одної або мають різні цілі виконання. Комунікація між процесами (inter-process communication, IPC) зазвичай

здійснюється через канали, пайпи, сокети або загальну пам’ять, і є значно складнішою та повільнішою, ніж взаємодія між потоками. Проте ізольованість процесів забезпечує кращу стабільність системи та безпечніше виконання критичних задач. У контексті ігрових додатків PLP може використовуватись, наприклад, для запуску окремого процесу аналітики, генерації звітів, збору телеметрії або обслуговування серверної частини гри.

Усі три види паралелізму не є взаємовиключними, а навпаки – часто доповнюють один одного. У складних програмних системах, таких як сучасні ігрові рушії, може одночасно реалізовуватись ILP на рівні апаратного забезпечення, TLP на рівні внутрішньої логіки гри та PLP – у частині обслуговування клієнтів або сервісів [4]. Застосування паралелізму на різних рівнях дозволяє розробникам максимально ефективно використовувати ресурси багатоядерних процесорів, підвищуючи швидкодію, зменшуючи затримки та забезпечуючи стабільну продуктивність навіть за умов високого навантаження.

У контексті паралельних обчислень важливим аспектом є оцінка ефективності масштабування продуктивності системи в залежності від кількості обчислювальних ядер або потоків. Для формального опису цієї залежності в науковій літературі широко використовуються дві ключові моделі: закон Амдала (Amdahl’s Law) та закон Густафсона (Gustafson’s Law). Обидва підходи дають змогу оцінити граничну продуктивність паралельної системи, однак виходять з різних припущень та слугують для різних цілей моделювання.

Закон Амдала був сформульований у 1967 році Ґене Амдалом і є класичною моделлю, яка описує обмеження на прискорення (speedup), що може бути досягнуто за допомогою паралелізації програми [5]. Він базується на припущенні, що певна частина програми (позначимо її як α) є строго послідовною і не може бути розпаралелена, а решта частини (1 – α) – може виконуватись паралельно.

Математично закон Амдала формулюється так:

$$S(n) = \frac{1}{\alpha + \frac{1-\alpha}{n}}$$

де:

– $S(n)$ – максимальне прискорення при використанні nnn процесорів,

– α – частка послідовного коду в загальній програмі ($0 \leq \alpha \leq 1$),

– n – кількість обчислювальних ядер.

Згідно з цим законом, навіть за великої кількості процесорів максимальне прискорення обмежується тим, наскільки значною є послідовна частина програми. Наприклад, якщо 10% програми не піддається паралелізації, то максимальне прискорення ніколи не перевищить 10×, навіть за ідеальної паралельної реалізації решти 90%.

Критика: Закон Амдала передбачає фіксований обсяг задачі (strong scaling), що часто не відповідає практиці, коли з ростом ресурсів обсяг задач також збільшується [6].

Альтернативний підхід запропонував Джон Густафсон у 1988 році. Його модель – закон Густафсона – виходить із припущення, що зі зростанням кількості процесорів збільшується і розмір задачі (weak scaling), тобто програма масштабуються відповідно до ресурсів. На відміну від Амдала, Густафсон фокусувався не на фіксованому обсязі роботи, а на збереженні часу виконання при зміні кількості процесорів.

Формула закону Густафсона:

$$S(n) = n - \alpha(n - 1)$$

де:

– $S(n)$– прискорення при nnn процесорах,

– α – частка послідовної частини програми,

– n – кількість ядер.

У цій моделі прискорення майже лінійно зростає зі збільшенням кількості ядер, що більше відповідає практичному досвіду в задачах на зразок обробки великих даних, симуляцій або складних ігрових сценаріїв. Закон Густафсона краще описує ситуацію, коли зростання обчислювальних ресурсів супроводжується збільшенням складності або обсягу задачі.

У контексті розробки ігрових додатків, обидва закони допомагають розробникам обґрунтувати доцільність використання паралельної обробки. Закон

Амдала акцентує увагу на необхідності мінімізувати послідовні ділянки коду, тоді як закон Густафсона відкриває можливості масштабування при збільшенні обсягу ігрової сцени, деталізації графіки або складності фізичних симуляцій [7].

Правильне застосування цих концепцій дозволяє оцінити межі ефективності розпаралелювання, виявити вузькі місця в архітектурі програмного забезпечення і обрати оптимальні стратегії реалізації багатопотоковості у високопродуктивних системах, зокрема в сучасних ігрових рушіях.

1.2 Архітектура багатоядерних процесорів: CPU vs GPU

Одним із ключових напрямів розвитку обчислювальних технологій є диференціація архітектур центральних процесорів (Central Processing Unit, CPU) та графічних процесорів (Graphics Processing Unit, GPU), які виконують різні функції у межах обчислювальної системи. Обидва типи процесорів підтримують багатоядерність, однак реалізують її за принципово різними підходами, що зумовлює відмінності у їхніх функціональних можливостях, ефективності виконання задач та застосуванні в ігрових додатках.

Central Processing Unit (CPU) – універсальним обчислювальним пристроєм загального призначення, оптимізованим для виконання послідовних інструкцій з високою швидкістю [8]. Сучасні CPU зазвичай мають від 4 до 32 високопродуктивних ядер, кожне з яких оснащене складними механізмами управління потоком інструкцій: багаторівнева кеш-пам’ять (L1–L3), механізми передбачення переходів, конвеєризація, out-of-order execution (рис. 1.1).



Рисунок 1.1 – Central Processing Unit

Переваги CPU:

– Висока продуктивність при обробці послідовних або слабо паралельних задач.

– Універсальність: підтримка широкого спектру інструкцій, складних умовних операторів, системного коду.

– Ефективна робота з великими обсягами оперативної пам’яті з низькою латентністю.

– Підтримка складної логіки та IO-операцій.

Недоліки CPU:

– Обмежена кількість ядер порівняно з GPU.

– Нижча ефективність при масово-паралельних обчисленнях.

– Менше одиниць обробки даних на одиницю площі кристалу [9].

Graphics Processing Unit (GPU), на відміну від CPU, GPU спочатку проєктувався для швидкої обробки графічних операцій, але з часом став потужною платформою для паралельних обчислень. Сучасні GPU складаються з сотень або тисяч простих ядер, які об'єднані в обчислювальні блоки (Streaming Multiprocessors, SM). Ця архітектура дозволяє обробляти велику кількість потоків одночасно (SIMD

– Single Instruction, Multiple Data), що ідеально підходить для задач із високим ступенем паралелізму: обробка зображень, фізичне моделювання, машинне навчання (рис. 1.2).



Рисунок 1.2 – Графічний процесор (GPU)

Переваги GPU:

– Величезна кількість ядер забезпечує масивний паралелізм.

– Висока пропускна здатність обчислень (TFLOPS).

– Ефективність при обробці однотипних операцій над великими масивами даних.

– Ідеальне середовище для реалізації векторних і матричних обчислень.

Недоліки GPU:

– Вища латентність пам’яті та складніша структура доступу до даних.

– Обмежена гнучкість: неефективні при складній логіці, умовах, циклах із розгалуженням.

– Потреба в спеціалізованих API (CUDA, OpenCL) або обгортках (наприклад, Aparapi для Java).

– Не завжди придатні для задач, де обсяг даних менший за обчислювальні витрати на передачу [10].

Зважаючи на фундаментальні архітектурні відмінності між CPU та GPU, доцільно подати їх порівняльну характеристику за ключовими технічними параметрами. Такий аналіз дозволяє глибше зрозуміти сфери ефективного застосування кожного з процесорів у межах ігрових застосунків, де важливими є як логічна обробка подій, так і високопродуктивне візуальне рендеринг. У табл. 1.1 наведено основні технічні та функціональні аспекти порівняння центральних і графічних процесорів.

Таблиця 1.1 – Порівняльна характеристика GPU та GPU

Критерій	**CPU**	**GPU**
Кількість ядер	4–32	1000+ (малі ядра)
Тип обчислень	Послідовні/розгалужені	Масово-паралельні (SIMD)
Гнучкість логіки	Висока	Низька (умовні розгалуження дорогоцінні)
Пропускна здатність	Середня	Висока
Затримка доступу до пам’яті	Низька	Вища
Споживання енергії	Помірне	Високе при повному навантаженні
Застосування в іграх	Ігрова логіка, ШІ, ІО	Рендеринг, фізика, обробка зображень

Сучасні центральні процесори (CPU), розроблені провідними компаніями Intel та AMD, демонструють високий рівень складності архітектури, орієнтованої на ефективне виконання паралельних обчислень, адаптацію до гібридних навантажень та оптимізацію енергоспоживання. Обидва виробники реалізують власні архітектурні підходи, однак спільними для них є такі риси, як використання багаторівневих кеш-пам’ятей, апаратна підтримка багатопотоковості, покращені засоби передбачення переходів, а також динамічне масштабування продуктивності.

Процесори Intel останніх поколінь, починаючи з архітектури Alder Lake, реалізують гібридний підхід до побудови багатоядерності. У межах однієї мікросхеми поєднуються два типи ядер: продуктивні (P-cores, Performance cores) та енергоефективні (E-cores, Efficient cores). Така архітектура дозволяє гнучко

розподіляти навантаження: важкі однопотокові або критичні задачі обробляються P-ядрами, тоді як фонові або паралельні потоки – E-ядрами. Крім того, Intel активно використовує технологію Hyper-Threading, яка дозволяє кожному ядру обробляти два потоки одночасно [11].

Серед додаткових переваг архітектури Intel:

– Багаторівневий кеш (L1, L2, L3) з високою пропускною здатністю;

– Передбачення переходів нового покоління, що зменшує кількість помилкових інструкцій у конвеєрі;

– Intel Thread Director – апаратний модуль, який у тандемі з ОС оптимізує розподіл задач між ядрами залежно від їхньої природи.

Процесори AMD, починаючи з архітектури Zen, також демонструють високий рівень багатоядерності та ефективності. Однією з ключових відмінностей є модульний принцип побудови на основі чиплетів (chiplets), де ядра організовані в блоки, з'єднані між собою через високошвидкісну шину Infinity Fabric. Такий підхід дозволяє масштабувати кількість ядер без значного збільшення витрат на виробництво, а також знижує теплове навантаження на окремі ділянки процесора.

Основні особливості архітектури AMD:

– Simultaneous Multi-Threading (SMT) – кожне ядро обробляє два потоки;

– Великий об’єм кеш-пам’яті третього рівня (до 64 МБ на CCD);

– Оптимізація для паралельних задач, зокрема в іграх, обробці відео та віртуалізації [12];

– Висока енергоефективність завдяки 5-нм і 7-нм техпроцесам (в архітектурах Zen 4 і Zen 3 відповідно).

І Intel, і AMD рухаються в напрямку збалансованої архітектури, яка поєднує високу тактову частоту з багатоядерністю, широкими шинами доступу до пам’яті та апаратною підтримкою векторних обчислень (AVX, AVX-512, FMA). Водночас Intel зосереджується на гібридних системах з динамічним управлінням потоками, а AMD – на модульності, масштабованості та високій щільності обчислювальних блоків.

У контексті розробки ігрових застосунків це означає, що архітектура CPU напряму впливає на ефективність розподілу ігрової логіки, обробку фізики, взаємодію з GPU, а також на оптимальність використання багатопотокових підходів у рушіях типу Unreal Engine чи Unity.

У сучасних багатоядерних процесорах реалізовано низку архітектурних рішень, які безпосередньо впливають на продуктивність обчислень у реальному часі. До таких ключових компонентів належать багаторівнева кеш-пам’ять, конвеєрне виконання інструкцій (pipeline), одночасна багатопотоковість (SMT) та нерівномірний доступ до пам’яті (NUMA). Усі ці елементи відіграють важливу роль у формуванні продуктивного середовища для виконання ігрових додатків [13].

Кеш-пам’ять – це надшвидкий тип пам’яті, який розташовується безпосередньо на кристалі процесора і слугує буфером між ядрами CPU та основною оперативною пам’яттю (RAM) (рис. 1.2). Вона поділяється на кілька рівнів:

– L1 – найменший за обсягом, але найшвидший (окремо для інструкцій та даних);

– L2 – більший і повільніший, часто спільний для ядра;

– L3 – найбільший обсяг, спільний для всіх ядер в одному модулі або чиплеті.



Рисунок 1.2 – Різниця в структурі CPU та GPU

Вплив на геймплей: ефективна кеш-підсистема знижує затримки при зверненні до пам’яті, що критично важливо для обробки внутрішньоігрових подій у режимі реального часу. Наприклад, збереження координат об’єктів, даних AI або

стану фізичних симуляцій у кеші дозволяє мінімізувати пропуски кадрів і забезпечити плавність анімації.

Pipeline – це архітектурна концепція, за якою інструкція розбивається на послідовні стадії (fetch, decode, execute, memory, writeback), які виконуються паралельно з інструкціями, що йдуть у черзі. Завдяки цьому одна інструкція виконується за кілька тактів, але різні стадії інших інструкцій обробляються одночасно.

Вплив на геймплей: pipeline дозволяє підвищити загальну пропускну здатність процесора. Це особливо важливо для частини ігрового рушія, яка виконує безперервні обчислення (наприклад, фізичні обчислення при зіткненнях чи обробка вводу гравця). Проте надто довгий pipeline (як у деяких Intel CPU) може призводити до penalty при неправильному прогнозуванні переходів, що негативно впливає на затримки.

Технологія SMT (у Intel – Hyper-Threading, у AMD – SMT) дозволяє одному фізичному ядру одночасно обробляти два логічні потоки. Це дає змогу краще використовувати обчислювальні ресурси ядра, особливо коли один із потоків простоює через затримку доступу до пам’яті [14].

Вплив на геймплей: SMT корисна для ігор, у яких велика кількість однотипних задач може бути розподілена між потоками (наприклад, багаточисельні NPC, частки, фонові процеси). Утім, у деяких випадках надмірне використання SMT може створювати конкуренцію за кеш-пам’ять та інші ресурси, що навпаки – погіршує продуктивність.

У багатопроцесорних або високоядерних системах пам’ять організована за моделлю NUMA, коли кожна група ядер (наприклад, на одному чиплеті) має свій локальний контролер пам’яті, до якого звертається швидше, ніж до віддалених блоків пам’яті іншої групи.

Вплив на геймплей: якщо ігровий рушій не враховує особливості NUMA, можливе зниження продуктивності через надмірні міжгрупові звернення до пам’яті. У добре оптимізованих системах, що використовують NUMA-стратегії (наприклад, CPU affinity), значно підвищується ефективність паралельного

виконання – особливо в мультиплатформенних або серверних ігрових застосунках (MMORPG, FPS з багатьма гравцями).

Архітектурні особливості CPU мають безпосередній вплив на загальну ефективність і плавність ігрового процесу. Оптимальне використання кеш-пам’яті, ефективне управління потоками через SMT, збалансоване використання pipeline та адаптація до NUMA-архітектури дозволяють максимально задіяти ресурси багатоядерних систем у реальному часі, забезпечуючи стабільну і якісну взаємодію з користувачем.

1.3 Підходи до програмування багатоядерних систем на Java

Одним із ключових елементів ефективного використання багатоядерних процесорів є правильне програмне управління потоками. Мова програмування Java пропонує широкий спектр інструментів для реалізації багатопотоковості, включаючи базові механізми Thread, високорівневі API Executors та фреймворк ForkJoinPool для обробки паралельних задач із розгалуженою структурою. Розуміння цих підходів і їх правильне застосування є необхідною умовою для створення масштабованих, продуктивних ігрових додатків [15].

Java Threads – клас Thread – це найпростіший спосіб створення окремого потоку виконання в Java. Він дозволяє визначити логіку виконання у методі run() та запускати її паралельно до основного потоку через метод start(). Такий підхід підходить для простих задач, але при збільшенні кількості потоків потребує додаткового управління життєвим циклом потоків і координації між ними.

Executors – фреймворк java.util.concurrent.Executors забезпечує високорівневу абстракцію над управлінням потоками. Замість створення нових потоків вручну, розробник створює пул потоків (наприклад, FixedThreadPool, CachedThreadPool), який динамічно розподіляє задачі, зменшуючи витрати на ініціалізацію кожного окремого потоку. Це особливо ефективно в іграх, де десятки дрібних завдань потребують асинхронної обробки: анімації, обробка подій, фізика, AI.

ForkJoinPool, для задач, що мають рекурсивну або деревоподібну структуру, у Java доступна реалізація фреймворку ForkJoinPool, який дозволяє ділити великі задачі на підзадачі (fork), обробляти їх паралельно та об’єднувати результати (join). Такий підхід добре працює для задач, пов’язаних із трасуванням променів, генерацією ландшафтів або симуляцією поведінки великої кількості NPC.

У багатопотокових програмах важливо забезпечити узгодженість доступу до спільних ресурсів, оскільки одночасне їх використання без контролю може призвести до race conditions, deadlocks або пошкоджених даних [16].

Synchronized, найбільш базовий механізм синхронізації в Java. Дозволяє оголосити критичну секцію, доступ до якої контролюється м’ютексом. Наприклад:

synchronized (sharedObject) { // безпечна робота з ресурсом}

ReentrantLock – пакет java.util.concurrent.locks пропонує більш гнучкі засоби синхронізації, зокрема ReentrantLock, який підтримує блокування з тайм-аутом, перевірку блокування, умовні змінні. Це особливо корисно в складних системах ігрової логіки.

Atomic змінні – клас java.util.concurrent.atomic містить атомарні обгортки (AtomicInteger, AtomicBoolean тощо), які дозволяють безпечно виконувати операції без блокувань, що зменшує накладні витрати і пришвидшує обчислення в простих конкурентних структурах.

Правильна організація багатопотокового виконання критично важлива для продуктивності ігрових додатків. З одного боку, багатопоточність дозволяє розділити задачі між ядрами CPU, тим самим зменшуючи навантаження на головний потік і забезпечуючи плавний геймплей. Наприклад, виділення окремих потоків для AI, фізики або обробки мережевого трафіку дозволяє уникнути затримок у головному графічному потоці, який відповідає за рендеринг.

Неправильне використання потоків може призвести до зворотного ефекту – перевантаження процесора, конфліктів за ресурси, надмірного контекстного перемикання або навіть блокувань (deadlocks). Тому при розробці ігор необхідно ретельно аналізувати доцільність багатопоточності, визначати оптимальну

кількість потоків з урахуванням кількості ядер CPU, а також мінімізувати взаємозалежності між потоками.

Ефективне використання інструментів багатопоточності в Java дозволяє створювати масштабовані, реактивні й високопродуктивні ігрові системи, здатні повною мірою використовувати переваги сучасних багатоядерних процесорів [17].

У сучасних ігрових рушіях, реалізованих мовою Java, багатопоточність використовується як основний механізм розподілу функціонального навантаження між різними компонентами системи. Основні задачі – рендеринг графіки, оновлення ігрової логіки та виконання фонових операцій – зазвичай розносяться по окремих потоках для досягнення стабільної продуктивності та уникнення блокувань у головному циклі гри. Типову структуру організації потоків у Java-гейменджині представлено на UML-діаграмі (рис. 1.3).

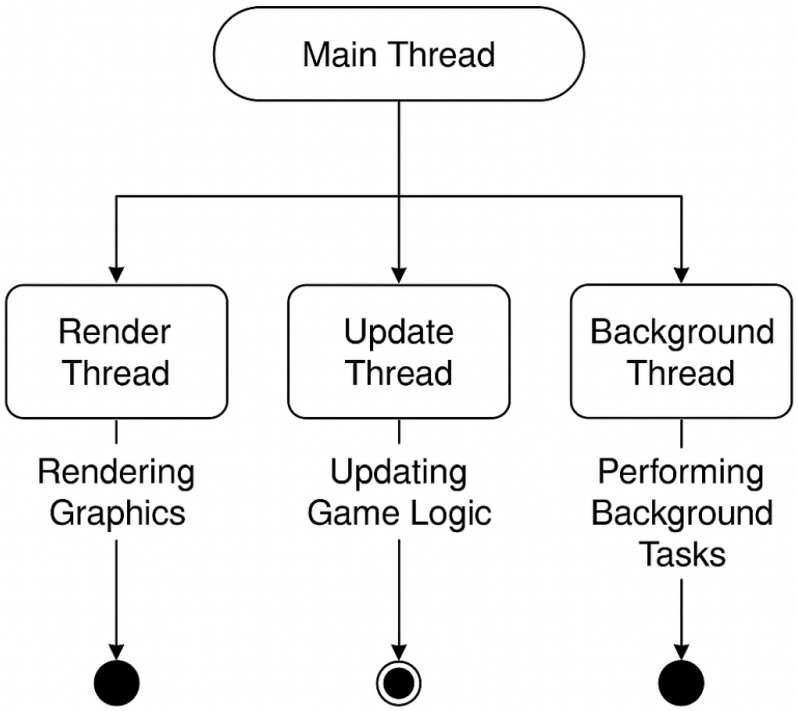


Рисунок 1.3 – UML-діаграма потокової архітектури Java-гейменджина: основні потоки обробки

1.4 Огляд технологій Java для високопродуктивних обчислень (Aparapi, TornadoVM)

Хоча Java традиційно вважається мовою загального призначення із високим

рівнем абстракції, упродовж останніх років вона активно адаптується до потреб високопродуктивних обчислень, зокрема на графічних процесорах (GPU). Поява спеціалізованих фреймворків, таких як Aparapi та TornadoVM, відкрила можливості інтеграції GPU-прискорення без необхідності переходу на мови нижчого рівня (наприклад, CUDA або OpenCL) [18]. Це особливо важливо в контексті ігрової індустрії, де ресурсоємні задачі, такі як фізичне моделювання, AI або масова обробка даних, потребують масштабованої обчислювальної потужності.

У середовищі Java GPU-обчислення реалізуються за допомогою таких платформ:

– Aparapi – Java API, який дозволяє автоматично компілювати частину Java-коду в OpenCL та виконувати його на GPU без ручного написання коду мовою C або C++.

– TornadoVM – віртуальна машина, яка інтегрується з GraalVM і дозволяє Java-програмам запускати частини коду на гетерогенних пристроях (CPU, GPU, FPGA) з автоматичним управлінням компіляцією і трансляцією в OpenCL, PTX або SPIR-V.

Обидві технології розширюють можливості використання Java у сценаріях, що вимагають високої паралельності.

Aparapi:

– підтримує написання високопродуктивного Java-коду безпосередньо, без потреби вручну писати OpenCL-кернели.

– мінімізує поріг входу для Java-розробників у GPU-програмування.

– автоматично виконує трансляцію Java-класів у OpenCL.

– підтримує fallback-режим: у разі відсутності GPU-контексту код виконується на CPU.

TornadoVM:

– більш гнучкий та масштабований, ніж Aparapi;

– підтримує багатомодульну компіляцію, аналіз потоку даних та оптимізацію на рівні графу задач;

– дозволяє запускати Java, Scala, Kotlin-код на GPU або FPGA;

– працює в рамках GraalVM і забезпечує адаптивне управління ресурсами в реальному часі.

Aparapi зручніший для початку роботи та простих задач, тоді як TornadoVM орієнтований на складні обчислювальні графи та масштабовані багатоплатформні проєкти.

У контексті ігрових додатків Aparapi та TornadoVM можуть застосовуватись для наступних задач:

– фізичне моделювання частинок – обчислення траєкторій, зіткнень або ефектів (наприклад, пил, вода, вибухи) значно пришвидшується при перенесенні на GPU;

– паралельне трасування променів (Ray Tracing) – TornadoVM дозволяє реалізувати масштабовані алгоритми рендерингу сцени з високою деталізацією;

– AI-агенти у великих симуляціях – використання Aparapi для одночасного виконання логіки десятків або сотень NPC;

– симуляції натовпу або динаміки поведінки – GPU використовується для обробки моделей руху, зіткнень і координації;

– фінансово-ігрові симулятори – обробка великої кількості однотипних транзакцій або параметрів (як у MMO-економіках).

Сучасна потреба у високопродуктивних обчисленнях вимагає використання не лише багатоядерних CPU, але й можливостей GPU-прискорення. У цьому контексті особливу роль відіграють інструменти, які дозволяють Java-розробникам реалізовувати паралельні обчислення на графічних процесорах без використання низькорівневих мов. Серед таких інструментів найбільш поширеними є Aparapi та TornadoVM, що надають різні рівні доступу до апаратного прискорення через інтеграцію з OpenCL, PTX або іншими платформами.

Для повного розуміння їх можливостей, доцільно здійснити порівняльний аналіз цих двох рішень за ключовими параметрами, що мають безпосередній вплив на ефективність, гнучкість і складність реалізації GPU-обчислень у Java-додатках, порівняння наведено в **табл. 1.2**.

Таблиця 1.2 – Порівняльна характеристика Aparapi та TornadoVM для GPU-обчислень у Java

Критерій	**Aparapi**	**TornadoVM**
Рік появи	~2011	~2019
Підхід до обчислень	Java → OpenCL	Java → GraalVM → OpenCL / PTX / SPIR-V
Підтримка GPU	Через OpenCL	Через OpenCL, PTX (NVIDIA), SPIR-V (Intel, AMD, Vulkan)
Мови програмування	Тільки Java	Java, Kotlin, Scala
Автоматична компіляція	Так (оголошені методи транслюються у OpenCL)	Так (автоматичне визначення обчислювальних графів)
Можливість fallback на CPU	Так	Так
Підтримка багатопристроєвих систем	Обмежена	Повна (CPU, GPU, FPGA одночасно)
Гнучкість налаштувань та оптимізацій	Середня	Висока (аналіз графу задач, профілювання під час виконання)
Сумісність із екосистемою Java	Висока	Висока (але потребує GraalVM)
Складність інтеграції	Низька (простий API)	Середня (потрібна конфігурація GraalVM, компілятора TornadoVM)
Сфера застосування	Просте GPU-прискорення (частки, NPC-логіка)	Складні гетерогенні обчислення (рендеринг, трасування променів, AI, тощо)

2 ПРОЄКТУВАННЯ ТА РОЗРОБКА ПРОГРАМНОГО МОДУЛЯ ДЛЯ ОБРОБКИ ІГРОВИХ ЗАДАЧ НА CPU

2.1 Вибір інструментів і бібліотек для реалізації багатопотоковості

Ефективна реалізація багатопотокової обробки задач у межах ігрового застосунку потребує ретельного вибору інструментів, що забезпечують стабільну продуктивність, масштабованість і простоту інтеграції. У межах даної роботи було обрано два основних підходи до реалізації паралельних обчислень: Java Concurrency API як основну платформу для багатопотоковості у високорівневому застосунку, а також OpenMP – як засіб організації паралелізму на рівні нативного коду з можливістю інтеграції через JNI (Java Native Interface) [19].

Бібліотека java.util.concurrent є частиною стандартної Java-платформи і надає широкий набір високорівневих засобів для управління потоками, чергами, семафорами, блокуваннями та пулом виконавців (ExecutorService). Її переваги:

– Гнучкість: підтримка фіксованих, динамічних та спеціалізованих пулів потоків;

– Безпечність: наявність потокобезпечних колекцій (наприклад, ConcurrentHashMap);

– Масштабованість: ефективна робота в багатоядерному середовищі;

– Зручна інтеграція: повна сумісність з JVM, без потреби у зовнішніх залежностях;

– Контроль життєвого циклу потоків: зручне завершення, зупинка та обробка виключень.

Ця бібліотека дозволяє розробнику сконцентруватися на логіці обробки, делегуючи керування ресурсами JVM.

OpenMP, для задач, що потребують максимального використання апаратних можливостей, зокрема CPU-bound обчислень, було досліджено можливість використання OpenMP – стандарту для паралельного програмування на C/C++ і Fortran. Через інтеграцію з Java (наприклад, JNI або JNA) певні обчислювальні

блоки можуть бути реалізовані в нативному коді з використанням директив OpenMP, що дозволяє:

– Розподіляти обчислення між усіма доступними ядрами процесора;

– Оптимізувати продуктивність для критичних секцій (наприклад, масові фізичні симуляції);

– Отримати приріст продуктивності до 2–4× у порівнянні з чистим Java-кодом (у деяких задачах).

У процесі розробки високопродуктивних програмних застосунків важливе значення має вибір відповідного інструменту або API для реалізації паралельних обчислень. Існує низка підходів, кожен з яких має свої сильні та слабкі сторони залежно від рівня абстракції, продуктивності, складності інтеграції та підтримуваної платформи. Нижче у табл. 2.1 наведено порівняльну характеристику найбільш поширених інструментів, що використовуються для створення багатопотокових і асинхронних програм.

Таблиця 2.1 Порівняння з альтернативами

Інструмент / API	**Рівень**	**Переваги**	**Недоліки**
Java Concurrency	Високий	Легко інтегрується, безпечний, універсальний	Менш оптимальний для важких обчислень
OpenMP + JNI	Низький / C++	Висока продуктивність, прямий контроль над ядрами	Ускладнена інтеграція, потреба в нативному коді
Akka (Scala)	Акторна модель	Масштабованість, зручна модель взаємодії об’єктів	Залежність від Scala, складна модель керування
Kotlin Coroutines	Асинхронність	Простіша обробка асинхронних задач, легка інтеграція в Android	Не підходить для складних CPU-bound задач
Reactive Streams (RxJava)	Потокові API	Зручно для реактивних інтерфейсів, IO-задач	Високий поріг входу, обмежене використання для CPU-обчислень

Враховуючи завдання проєкту – реалізацію багатопоточних механізмів у Java-застосунку, орієнтованому на обробку ігрових задач, – вибір Java Concurrency

API є найбільш виправданим. Він поєднує зручність**,** захищеність і достатню продуктивність для більшості задач, що не потребують ручного управління апаратними ресурсами. У випадках, де критичною є максимізація продуктивності (наприклад, великі симуляції чи генерація карт), можлива інтеграція обчислювальних блоків на C++ з використанням OpenMP, що дозволяє ефективно масштабувати обчислення в середовищі багатоядерного процесора.

2.2 Архітектура і структура розробленої системи

У межах реалізації багатопоточної обробки в ігровому застосунку було спроєктовано модульну архітектуру, яка забезпечує розподіл функціонального навантаження між незалежними компонентами системи. Такий підхід дозволяє ефективно масштабувати застосунок, ізолювати критичні секції, а також оптимізувати використання обчислювальних ресурсів процесора шляхом паралельного виконання задач.

Розроблений програмний модуль поділяється на низку логічно відокремлених функціональних компонентів, кожен з яких реалізує певну підсистему ігрового рушія:

– Core Engine (Ядро гри) – реалізує основний цикл гри (game loop), обробку кадрів, синхронізацію між компонентами [20].

– Rendering Module (Рендеринг) – відповідає за виведення графічної інформації на екран. Цей компонент працює в окремому потоці для забезпечення високої частоти кадрів.

– Physics Module (Фізичний рушій) – виконує симуляцію фізичних процесів, таких як гравітація, зіткнення, рух об'єктів. Реалізується з використанням багатоядерності (через ForkJoinPool або OpenMP).

– AI Module (Штучний інтелект) – обробляє логіку поведінки NPC (неігрових персонажів), приймає рішення на основі стану гри, працює паралельно в окремому потоці.

– Input Handler – приймає сигнали від користувача та передає їх у Core Engine.

– Resource Manager – відповідає за завантаження текстур, моделей, звуків; реалізує кешування та асинхронний доступ до ресурсів.

Для організації коду застосовано (рис. 2.1) об'єктно-орієнтований підхід, згідно з яким система поділена на такі пакети (packages).

```
com.gameengine
|
├── core              → ядро рушія, керування циклами
├── graphics          → рендеринг, буфери, OpenGL-виклики
├── physics           → обробка зіткнень, моделювання руху
├── ai                → логіка NPC, дерево рішень
├── input             → клавіатура, миша, ігрові контролери
├── resources         → менеджер ресурсів, лоадери
├── utils             → допоміжні класи: логування, профілювання
└── native            → інтеграція з OpenMP через JNI (опціонально)
```

Рисунок 2.1 – Модульна структура програми

Для координації між потоками використовується ExecutorService (пул фонових потоків), а також thread-safe структури (ConcurrentLinkedQueue, BlockingQueue) для обміну повідомленнями між модулями [21]. Синхронізація станів здійснюється через ReentrantLock або synchronized-блоки в критичних секціях (рис. 2.2).

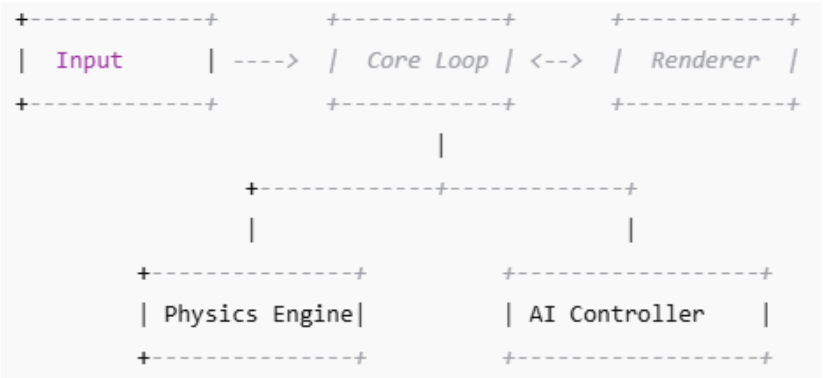


Рисунок 2.2 – Схема модульної архітектури ігрової системи з багатопоточністю

Для кращого розуміння внутрішньої структури та взаємодії компонентів розробленої системи доцільно представити UML-діаграму класів, яка ілюструє ключові модулі, їхні методи та зв’язки між ними [22]. Архітектура базується на принципах об’єктно-орієнтованого проєктування, що забезпечує модульність, повторне використання коду та зручність масштабування. Кожен клас відповідає за окрему функціональну підсистему, а їх взаємодія координується ядром рушія (рис. 2.3).

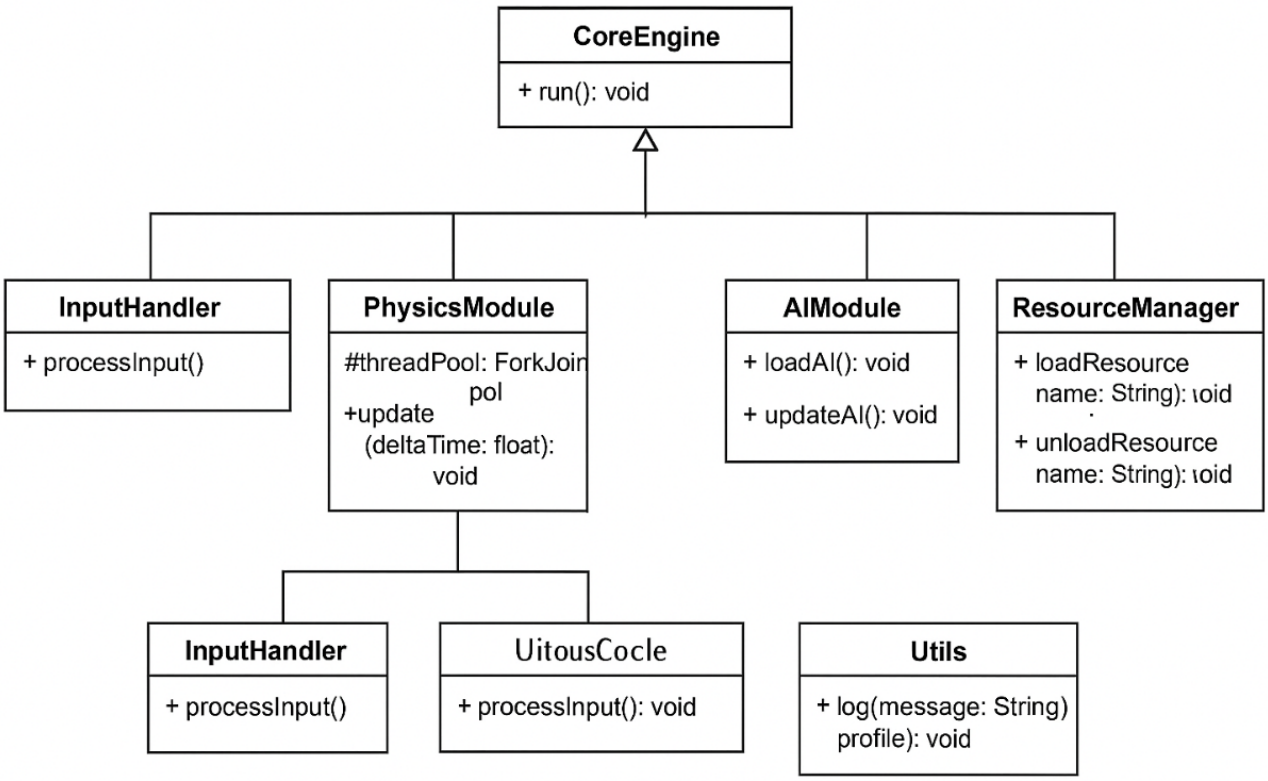


Рисунок 2.3 – UML-діаграма класів ігрового застосунку з підтримкою багатопоточності

Підтримка багатопоточності в розробленій системі реалізована шляхом організації незалежного виконання ключових модулів у фонових потоках, що дозволяє ефективно задіяти ресурси багатоядерного процесора. Основою для управління паралельними задачами слугує стандартний механізм ExecutorService з фіксованим пулом потоків, який забезпечує планування та контроль за виконанням задач фізичної симуляції, логіки NPC, обробки ресурсів тощо. Для підвищення ефективності в задачах, що добре масштабуються, таких як обробка великої

кількості фізичних об'єктів, використовується високорівнева модель ForkJoinPool, яка автоматично розподіляє підзадачі між усіма доступними ядрами CPU [23].

Асинхронне завантаження ресурсів реалізовано за допомогою механізмів, які дозволяють уникнути блокування основного потоку та забезпечити безперервність ігрового процесу. Водночас розподіл потоків здійснено за логічними підсистемами: потік рендерингу відповідає за графіку, оновлювальний потік – за фізику та ігрову логіку, а введення обробляється незалежно для мінімізації затримок реакції на дії користувача.

Для забезпечення безпечної взаємодії між потоками застосовуються потокобезпечні структури даних (BlockingQueue, ConcurrentHashMap) та механізми синхронізації (ReentrantLock, AtomicInteger), які дозволяють уникнути конфліктів при спільному доступі до ресурсів. Такий підхід забезпечує не лише стабільність системи під навантаженням, а й гнучкість у розширенні функціональності без шкоди для продуктивності.

2.3 Інтеграція та взаємодія потоків у задачах обробки ігрової логіки

У багатопотоковій архітектурі ігрових застосунків особливе значення має не лише розподіл задач між потоками, а й ефективна взаємодія між ними. У розробленій системі реалізовано поділ основних підсистем на окремі потоки, що дозволяє виконувати обчислення паралельно та зменшити загальне навантаження на центральний цикл гри. Такий підхід сприяє підвищенню продуктивності, забезпеченню стабільного FPS і зниженню затримок у взаємодії з користувачем.

Зокрема, фізична симуляція виконується в окремому потоці, який обробляє взаємодію об'єктів, зіткнення та оновлення позицій на основі фізичних параметрів сцени. Ігрова логіка, пов’язана з діями гравця, виконанням сценаріїв, а також внутрішньоігровими подіями, обробляється в іншому потоці. Такий розподіл дає змогу уникнути перевантаження головного потоку, відповідального за рендеринг і оновлення графічного інтерфейсу, що особливо важливо в умовах складних або динамічних сцен.

Для координації між потоками застосовуються потокобезпечні структури даних, зокрема BlockingQueue для передачі подій між логічним і фізичним модулями, та ConcurrentHashMap для зберігання глобального стану об'єктів. У критичних ділянках, де можлива конкуренція за ресурси, використовується ReentrantLock, а для простих операцій над змінними – AtomicInteger та інші класи з пакету java.util.concurrent.atomic [24].

Разом із перевагами, багатопоточна взаємодія має і потенційні ризики. Однією з поширених проблем є гонки даних (race conditions) – ситуації, коли кілька потоків одночасно змінюють спільну змінну без належної синхронізації, що призводить до непередбачуваної поведінки. Ще однією загрозою є взаємні блокування (deadlocks), які виникають у разі циклічної залежності потоків один від одного при захопленні ресурсів. Для уникнення цих проблем у системі застосовано принцип мінімізації спільного доступу до ресурсів та використання немодифіковних (immutable) об'єктів у міжпотоковій взаємодії.

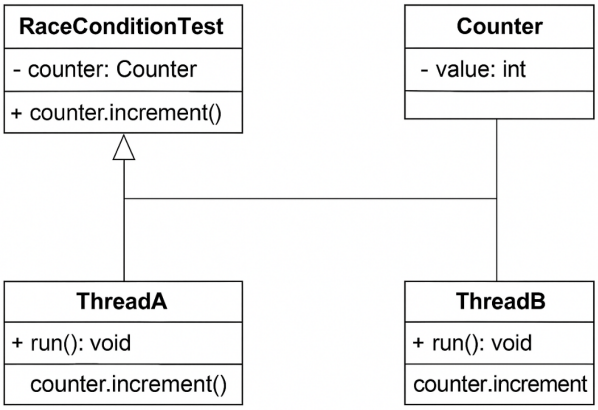


Рисунок 2.4 – UML-діаграма демонстраційного прикладу виникнення race condition у багатопоточному середовищі Java

Інтеграція і взаємодія потоків у системі організована за принципами асинхронної обробки, контрольованої синхронізації та ізоляції відповідальностей, що забезпечує не лише високу продуктивність, а й стабільну, передбачувану роботу застосунку в умовах паралельного виконання численних задач.

2.4 Оптимізація навантаження на ядра процесора

Раціональне використання обчислювальних ресурсів багатоядерного процесора є ключовим чинником досягнення стабільної продуктивності ігрового застосунку, особливо за умов паралельного виконання численних задач. Оптимізація навантаження передбачає як ефективний розподіл задач між ядрами, так і моніторинг їхнього виконання з подальшим удосконаленням обчислювальної стратегії.

У межах реалізованої архітектури задачі розподіляються за логічними доменами, що дозволяє уникати конфліктів і підвищити ефективність кешування. Наприклад, один потік відповідає за обробку фізики, інший – за оновлення AI, третій – за відтворення графіки. Це дозволяє кожному ядру обслуговувати окрему ділянку ігрового процесу без надлишкової синхронізації.

Крім статичного розподілу, застосовуються динамічні алгоритми, зокрема робота через пул потоків **(**ExecutorService**)** із фіксованою або адаптивною кількістю воркерів [25]. Такий підхід дозволяє системі автоматично балансувати навантаження залежно від поточного стану виконання задач.

У випадках із великим обсягом однотипних обчислень, наприклад обробки списку об'єктів сцени, застосовується ForkJoinPool, який реалізує рекурсивний поділ задач (work stealing algorithm) – потоки, що простоюють, "викрадають" задачі з черг інших потоків, забезпечуючи повне завантаження всіх ядер.

Вимірювання та оптимізація навантаження, для виявлення вузьких місць у продуктивності використовуються профайлери, які дають змогу оцінити:

– завантаження ядер процесора в режимі реального часу;

– час виконання кожного потоку та його блокування;

– параметри GC (збірника сміття), які часто впливають на стабільність FPS.

Оптимізація здійснюється шляхом:

– зменшення тривалості критичних секцій (коротші synchronized-блоки);

– обмеження кількості активних потоків до кількості логічних ядер CPU;

– розбиття великих задач на дрібні підзадачі, які обробляються паралельно;

– кешування часто використовуваних даних у локальних структурах (наприклад, ThreadLocal).

Для аналізу продуктивності було використано такі інструменти:

Java VisualVM – стандартний профайлер, який дозволяє відслідковувати використання пам’яті, активність потоків, виклики методів. У процесі тестування виявлено, що фізичний модуль мав найвище навантаження, що дало змогу виділити йому окремий пул потоків [26].

JMH (Java Microbenchmark Harness) – інструмент для мікробенчмарків, який використовувався для точного вимірювання часу виконання окремих методів (наприклад, updatePhysics()), оцінки середнього часу виконання (average time) та коефіцієнта відхилення [27].

На основі зібраних даних було виконано оптимізацію – скорочено обсяг синхронізованих блоків, спрощено логіку колізій, та виділено частину обчислень у окремі фонові завдання (рис. 2.5).

```
@Benchmark
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public void testPhysicsUpdate() {
    physicsModule.update(0.016f);
}
```

Рисунок 2.5 – Приклад використання JMH для бенчмарку методу оновлення фізики у грі

Оптимізація навантаження на ядра процесора досягається шляхом грамотного розподілу обчислювальних задач, адаптивного керування потоками та використання профілювання для постійного моніторингу й удосконалення продуктивності. Це дозволяє досягти стабільної роботи застосунку навіть за високого рівня паралелізму та складності обробки ігрової логіки.

3 ТЕСТУВАННЯ ЕФЕКТИВНОСТІ ТА ОЦІНКА ПРОДУКТИВНОСТІ ЗАПРОПОНОВАНОГО РІШЕННЯ

3.1 Методика тестування: критерії, середовище, сценарії

У даному підрозділі представлено підхід до оцінки ефективності функціонування програмного модуля, розробленого для обробки ігрових задач у багатоядерному середовищі. Оцінювання включає як функціональні, так і продуктивні аспекти реалізації системи з використанням паралельного програмування на базі CPU [28].

Оцінка продуктивності здійснюється на основі функціонально-вартісного аналізу (ФВА) – методології, яка дозволяє визначити обґрунтованість обраних технічних рішень з точки зору їхнього впливу на ресурси системи. Такий підхід є доцільним у контексті ігрової індустрії, де критично важливо забезпечити оптимальний розподіл навантаження на ядра процесора без втрати якості або швидкодії.

Методика ФВА починається з ідентифікації основних функцій, що виконує система: генерація ігрової сцени, обробка дій користувача, рендеринг кадру, обчислення колізій тощо. Усі функції класифікуються на ті, що мають прямий вплив на продуктивність системи, і ті, що є допоміжними. У подальшому допоміжні функції або оптимізуються, або реорганізуються задля зменшення споживання ресурсів [28].

Для кожної ключової функції фіксується середній час виконання, кількість використаних потоків, а також ступінь навантаження на ядра процесора. На основі цих даних формується профіль продуктивності застосунку. Також оцінюється використання кешу, ефективність розподілу задач між потоками, та поведінка програми в умовах конкурентного доступу до ресурсів.

Сценарії тестування передбачають як штучні контрольовані умови (однотипні послідовні дії в ігровій сцені), так і реалістичні сценарії взаємодії, що імітують поведінку користувача в реальному часі. Це дозволяє виявити вузькі місця

та оцінити масштабованість рішення при зростанні навантаження.

Метод функціонально-вартісного аналізу (ФВА) було застосовано для проведення техніко-економічного обґрунтування розробки програмної системи для обробки ігрових задач з використанням багатоядерних процесорів. Оскільки ключові архітектурні рішення мають системний характер, кожен функціональний компонент застосунку має відповідати загальним вимогам до ефективності, масштабованості та продуктивності.

Фактично техніко-економічний аналіз у цьому випадку полягає в оцінці функцій програмного продукту, спрямованого на забезпечення максимальної продуктивності в умовах обмежених ресурсів типового користувацького ПК.

До розробленого програмного продукту висуваються такі технічні вимоги:

1. Сумісність з персональними комп’ютерами із типовою багатоядерною архітектурою (від 4-х ядер і вище) [28, 29].
2. Висока швидкодія при обробці даних і виконанні паралельних ігрових сценаріїв.
3. Простий, інтуїтивно зрозумілий інтерфейс взаємодії.
4. Гнучке налаштування кількості потоків та конфігурації системи.
5. Мінімальні витрати на впровадження та розгортання у середовищі кінцевого користувача.

Базова функція F0 — реалізація ефективного програмного модуля, здатного до обробки ігрових даних у паралельному режимі з використанням багатоядерного процесора.

Виходячи з мети дослідження, можна виділити такі основні функції продукту:

- F1 – вибір мови програмування;
- F2 – вибір фреймворку або бібліотеки для реалізації багатопотоковості;
- F3 – вибір операційної системи, що підтримує багатоядерне розгортання.

Кожна функція має кілька варіантів реалізації, серед яких:

F1: вибір мови програмування

1. Java

2. Python
3. C++

F2: вибір бібліотеки/підходу до паралельного програмування

1. Java Threads
2. Fork/Join Framework
3. Parallel Streams

F3: вибір ОС

1. Windows
2. Linux

Варіанти реалізації основних функцій програмного продукту наведено у морфологічній карті системи (див. рис. 4.1). Морфологічна карта дає змогу наочно представити всі можливі комбінації реалізацій ключових функцій, що забезпечують повноцінну роботу програмного забезпечення. Кожна функція (F1–F3) має кілька альтернатив реалізації, які можуть комбінуватись між собою для досягнення оптимального варіанту системи з точки зору технічної ефективності та витрат [30].

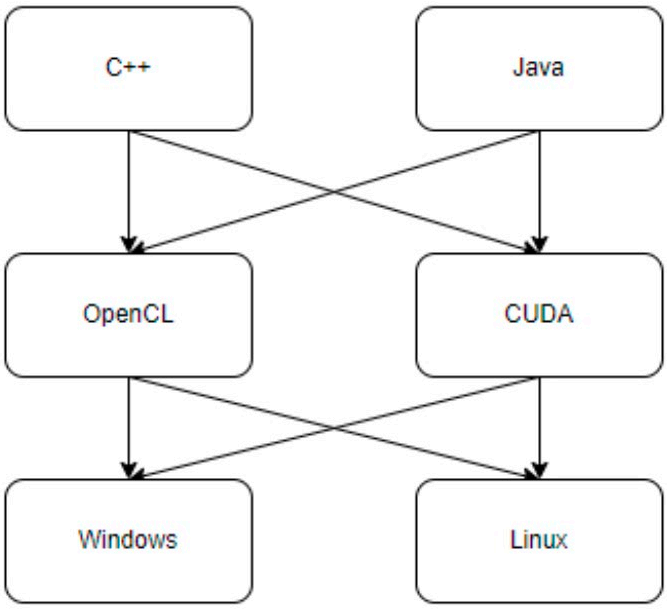


Рисунок 3.1 – Морфологічна карта варіантів реалізації функцій

Таблиця 3.1 – Порівняння варіантів реалізації функцій програмного продукту

Функція	Варіант реалізації	Переваги	Недоліки
F_1 – Мова програмування	A – C++	Висока продуктивність, ближчий контроль над апаратними ресурсами	Складність керування пам’яттю, вища вартість розробки
	B – Java	Зручність розробки, кросплатформеність, багатий набір бібліотек	Обмежений доступ до низькорівневих ресурсів GPU
F_2 – API для багатопотоковості	A – OpenCL	Незалежність від виробника GPU, підтримка різних платформ	Нижча оптимізація, складніший синтаксис
	B – CUDA	Висока оптимізація для Nvidia GPU, ефективна реалізація паралельних алгоритмів	Прив’язка до апаратного забезпечення Nvidia
F_3 – Операційна система	A – Windows	Поширеність, сумісність з великою кількістю ПЗ	Вартість ліцензії, складність кастомізації
	B – Linux	Безкоштовна, зручні пакетні менеджери, гнучкість налаштування	Може бути несумісною з деякими офісними пакетами (Microsoft Office)

На основі аналізу переваг і недоліків кожного з варіантів реалізації функцій, поданих у таблиці, було визначено доцільність відбору оптимальних комбінацій для реалізації програмного продукту. Варіанти, що не відповідають сучасним вимогам до універсальності, продуктивності або зручності розгортання, позначені як менш доцільні та виключені з подальшого розгляду [30-32].

Функція F_1 – Мова програмування:

Оскільки однією з ключових вимог до системи є кросплатформеність, легкість у розробці, а також наявність відповідного бібліотечного забезпечення для роботи з потоками, вибір зупинено на мові Java (варіант Б). Варіант А (C++) було відкинуто через складність пам'яттєвого керування та вищу вартість підтримки коду [31].

Функції F_2 (API паралельних обчислень) та F_3 (Операційна система):

Оскільки як OpenCL, так і CUDA можуть бути реалізовані для роботи з Java через відповідні бібліотеки (JCuda, Aparapi, TornadoVM), до подальшого аналізу було включено обидва варіанти. Аналогічно, і Windows, і Linux можуть забезпечити необхідне середовище для запуску системи, тому обидві ОС залишаються у розгляді.

Таким чином, було сформовано чотири допустимі комбінації реалізації системи:

1. F_1(Б) – F_2(Б) – F_3(Б) (Java, CUDA, Linux)
2. F_1(Б) – F_2(А) – F_3(Б) (Java, OpenCL, Linux)
3. F_1(Б) – F_2(А) – F_3(А) (Java, OpenCL, Windows)
4. F_1(Б) – F_2(Б) – F_3(А) (Java, CUDA, Windows)

У подальшому ці варіанти будуть оцінені за системою параметрів, що включає продуктивність, масштабованість, вартість впровадження та зручність підтримки. Це дозволить вибрати оптимальну конфігурацію для реалізації системи в контексті сучасних ігрових задач і паралельної обробки даних [32].

3.2 Порівняльний аналіз продуктивності (багатоядерність vs однопоточність)

Виклик методу execute() для об'єкта TaskSchedule у TornadoVM ініціює повний цикл обробки: від створення буферів OpenCL до компіляції байт-коду Java у код OpenCL та подальшого його виконання на обраному цільовому пристрої (наприклад, GPU або інший прискорювач). TornadoVM виконує автоматичну генерацію графу завдань, у межах якого методи об'єднуються в єдиний блок

компіляції.

Ця особливість дозволяє значно зменшити накладні витрати на передачу даних між центральним процесором (хостом) та прискорювачем. Така оптимізація особливо важлива в системах, де хост і пристрій не мають спільного доступу до оперативної пам’яті — типовий випадок для дискретних графічних процесорів. Проте у випадку з інтегрованими рішеннями, такими як AMD APU, ARM Mali або Intel HD Graphics, обмін даними може відбуватись ефективніше завдяки спільному пулу пам’яті, що додатково підвищує продуктивність [33].

```
1.class Compute {
2.   public static void matrixMultiplication(final float[] A,
   final float[] B, final float[] C, final int size) {
3.        for (@Parallel int i = 0; i < size; i++) {
4.            for (@Parallel int j = 0; j < size; j++) {
5.                float sum = 0.0f;
6.                for (int k = 0; k < size; k++)
7.                    sum += A[(i * size) + k] * B[(k * size) + j];
8.                C[(i * size) + j] = sum;
9.            }
10.         }
11.    }
12.}
```

Для аналізу ефективності застосування TornadoVM у задачах паралельних обчислень було проведено експеримент із виконанням алгоритму множення матриць розміром 512×512. Застосування відбувалося на графічному процесорі Nvidia GTX 1050 упродовж 100 ітерацій, з метою усереднення результатів. Результати виконання із використанням TornadoVM:

Час виконання (ітерація 1): 77568790 нс (0.0776 с)

Час виконання (ітерація 2): 3133182 нс (0.0031 с)

Час виконання (ітерація 3): 3126146 нс (0.0031 с)

Як видно, перша ітерація займає значно більше часу, що зумовлено процесом JIT-компіляції, який виконується один раз під час першого запуску TaskSchedule. Після генерації OpenCL-коду з Java-байткоду, TornadoVM зберігає згенеровані двійкові файли у кеші, що дозволяє уникнути повторної компіляції при наступних запусках. Для порівняння була виконана послідовна реалізація алгоритму множення матриць на Java без використання TornadoVM:

Час виконання (ітерація 1): 259398036 нс (0.2594 с)

Час виконання (ітерація 2): 247857535 нс (0.2479 с)

Перша ітерація з TornadoVM була у 3.3 раза швидшою за послідовну реалізацію на Java.

Починаючи з другої ітерації, прискорення склало понад 80 разів.

Під час тестування із використанням Java JMH Benchmark Framework спостерігалося середнє прискорення:

- до 3.6x на багатоядерному CPU (у порівнянні з Java HotSpot),
- до 39x на GPU Intel HD Graphics,
- до 270x на GPU Nvidia GTX 1050.

(Рис. 3.2) демонструє графік прискорення TornadoVM на різних пристроях OpenCL у логарифмічному масштабі [33, 34].

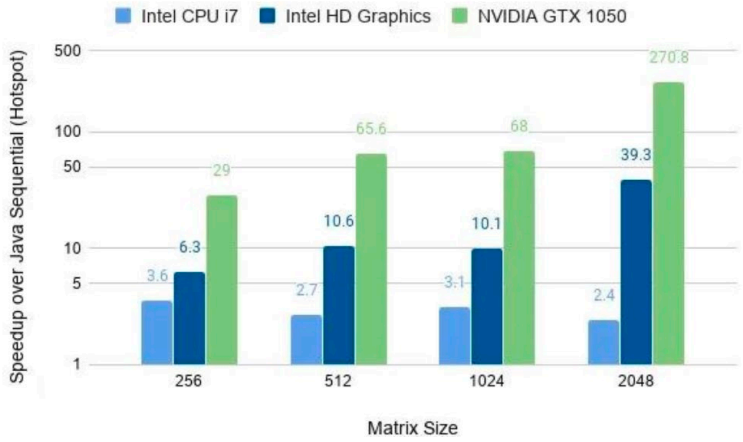


Рисунок 3.2 – Порівняльна швидкодія TornadoVM на різних пристроях при множенні матриць різного розміру

Представлено результати дослідження продуктивності JIT-компілятора, який забезпечує компіляцію та оптимізацію паралельного Java-коду для виконання на графічних процесорах. Тестування проводилось на платформі з двома 10-ядерними процесорами IBM POWER8 (3.69 ГГц, 256 ГБ оперативної пам’яті), що підтримують до 160 апаратних потоків завдяки SMT, а також GPU NVIDIA Tesla K40m із 2880 ядрами CUDA та 12 ГБ глобальної пам’яті. Платформа працювала під керуванням ОС Ubuntu 14.10 з підтримкою CUDA 5.5.

Компілятор реалізовано на базі IBM Testarossa JIT у складі IBM SDK Java Technology Edition (версія 8) із впровадженням чотирьох ключових оптимізацій, які

не потребують втручання користувача. Оптимізації включають покращене управління пам’яттю, мінімізацію обсягу переданих даних між хостом і GPU, інтенсивніше використання кешу, а також підтримку специфічних особливостей Java, таких як обробка винятків і виклики віртуальних методів усередині GPU-ядер.

Середовище виконання Java на основі динамічного профілювання визначає, які методи варто компілювати. Після трансляції байт-коду Java у проміжне представлення JIT-компілятор застосовує низку класичних оптимізацій, як-от усунення мертвого коду, поширення копіювання, часткове усунення надмірностей. Ці оптимізації були запозичені з практик для CPU, але адаптовані для гетерогенних обчислювальних пристроїв [35].

У табл. 3.2 наведено перелік еталонних задач, використаних для оцінки ефективності обчислень на графічних процесорах із різними розмірами даних та типами навантаження.

Таблиця 3.2 – Набір еталонних задач для оцінювання продуктивності системи GPU-обчислень

Назва алгоритму	Призначення	Розмірність задачі	Тип даних
OptionCalc	Модель розрахунку вартості європейських опціонів (call/put) у фінансових обчисленнях	4 194 304 віртуальні опціони	double
SecureCrypt	Криптографічна обробка даних із набору JGB (Java Grande Benchmarks)	N = 50 000 000	int

Продовження таблиці 3.2

SparseMM	Розріджене множення матриць (Sparse Matrix	N = 500 000	double

	Multiplication)		
FunctionSeries	Обчислення ряду функцій із JGB (наприклад, розклад у ряд Тейлора)	N = 1 000 000	double
MRI_Recon	Моделювання тривимірної реконструкції медичних зображень	64×64×64	float
DenseMM	Базове множення щільних матриць A × B	1024×1024	double
GEMM_Java	Множення: D = αAB + βC з бібліотеки PolyBench	1024×1024	int
GESUMMV_Java	Комбіновані операції скалярного, векторного та матричного множення (PolyBench)	4096×4096	int

Для оцінки продуктивності реалізованих рішень було використано вісім еталонних тестів, представлених у таблиці 3.2.1. Кожне завдання виконувалося у двох варіантах: послідовній реалізації на Java та паралельній, побудованій на використанні потоків із лямбда-виразами. Паралельні потоки в цьому випадку були реалізовані засобами фреймворку fork/join, що забезпечує ефективну роботу багатопотокових задач у середовищі JVM, а також із можливістю перенаправлення обчислень на графічні процесори (GPU) [36].

Паралельне виконання проводилось у двох конфігураціях:

– 160 потоків CPU: виконання здійснювалося на CPU за допомогою Java fork/join, де максимальна кількість потоків використовувалась автоматично (160 логічних потоків на платформі з IBM POWER8).

– GPU-конфігурація: запуск здійснювався на графічному процесорі (NVIDIA Tesla K40m) з використанням генерації коду для CUDA. Кожне ядро GPU запускалося з розподіленням у 1024 потоки на блок.

Замір ефективності проводився за допомогою системного виклику `System.nanoTime()`, який забезпечує точне визначення тривалості виконання кожного етапу. Ураховувалися всі витрати на ініціалізацію, зокрема: накладні витрати fork/join (для CPU) і виклики API драйвера CUDA, включно з виділенням пам’яті та передачею даних (для GPU).

Кожен тест запускався 30 разів в межах одного сеансу JVM, після чого середній час останніх 10 виконань використовувався як показник ефективності в усталеному режимі. Це дозволило мінімізувати вплив першочергових накладних витрат, зокрема через «розігрів» JIT-компілятора [37].

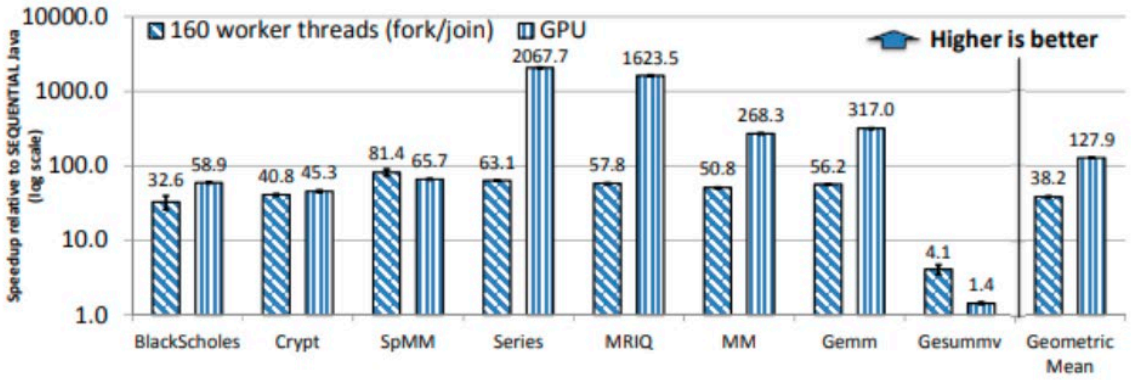


Рисунок 3.3 – Прискорення виконання Java-коду на GPU у порівнянні з CPU

Як показано на рис. 3.3, максимальні значення прискорення розраховані відносно базової послідовної реалізації Java-програми. Найвище прискорення для конфігурації з 160 потоками на CPU (використовуючи Java fork/join) досягло 81,4× у задачі SpMM (Sparse Matrix Multiplication). У свою чергу, для виконання на GPU найбільше прискорення зафіксовано для задачі Series — воно склало 2067,7×.

Цікаво, що у випадках SpMM та Gesummv продуктивність на CPU була вищою, ніж на GPU, що зумовлено значними витратами на обмін даними між хостом та відеокартою, які особливо впливають на розріджені або малі задачі.

Узагальнено, за середнім геометричним прискоренням, GPU забезпечив

покращення у 127,9 разів порівняно з послідовною Java-реалізацією, та у 3,3 рази — у порівнянні з багатопотоковим виконанням на CPU з використанням 160 апаратних потоків.

Граничні (мінімальні, середні та максимальні) значення параметрів програмного продукту визначаються з урахуванням вимог замовника та умов експлуатації системи, як показано нижче. У табл. 3.3 наведено перелік основних параметрів, що враховуються під час техніко-економічної оцінки програмного продукту [35-37].

Таблиця 3.3 – Оціночні параметри ефективності програмного рішення

Показник продуктивності	Позначення	Одиниця вимірювання	Гірше значення	Середнє значення	Краще значення
Максимальний обсяг оперативної пам’яті	X1	Гігабайт	16	8	4
Середній час виконання обчислювальної задачі	X2	Секунди	35	22	9
Обсяг коду системного модуля (оцінка)	X3	Рядки коду	500	220	130

На основі даних, наведених у таблиці 3.3, були побудовані графічні залежності, що відображають зміну основних параметрів ефективності програмного продукту при різних сценаріях експлуатації [37].

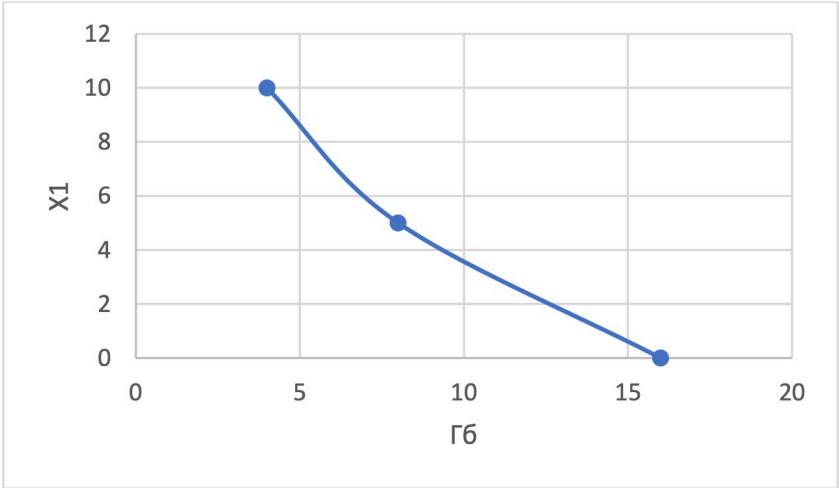


Рисунок 3.4 – Об’єм пам’яті, необхідний при різних конфігураціях системи

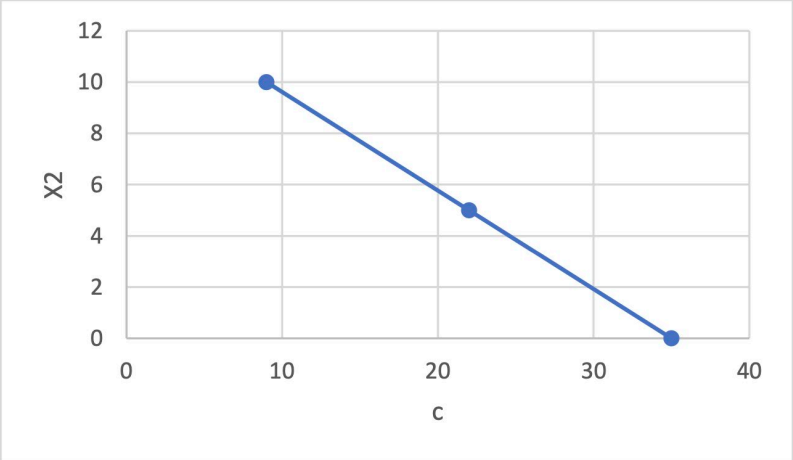


Рисунок 3.5 – Динаміка часу виконання задачі залежно від конфігурації

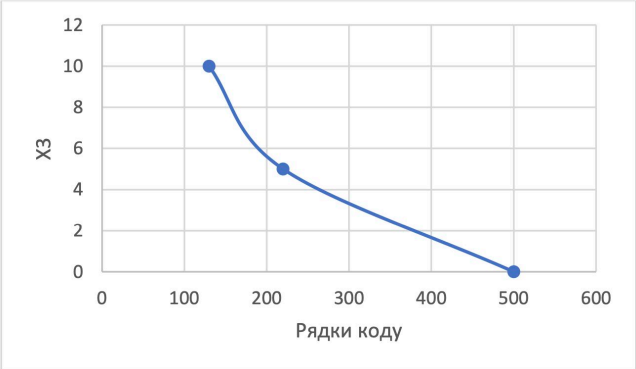


Рисунок 3.6 – Порівняння оцінки складності коду за кількістю рядків

3.3 Аналіз стабільності, масштабованості та виявлених обмежень

Після проведення детального обговорення й аналізу кожен експерт надає оцінку ступеню важливості параметрів у контексті поставленої мети — розробки програмного продукту з максимально зручним інтерфейсом та зрозумілою взаємодією з користувачем [38].

Значущість кожного з параметрів визначається із застосуванням методу попарного порівняння, що дозволяє встановити відносні коефіцієнти важливості. Оцінювання виконувалось експертною групою у складі 7 осіб.

Процедура визначення вагових коефіцієнтів складалася з таких етапів:

1. Присвоєння кожному параметру рангу відповідно до його значущості.
2. Перевірка узгодженості експертних оцінок і придатності до подальшого аналізу.
3. Побудова матриці попарного порівняння та обчислення коефіцієнтів пріоритету.
4. Нормалізація результатів і розрахунок фінального коефіцієнта значущості для кожного параметра.

Результати експертного ранжування представлені у табл. 3.4.

Таблиця 3.4 – Результати ранжування параметрів за методом попарного порівняння

	Параметр	Сума рангів	Відхилення Δi	Δi^2
1	X1	20	-6	36
2	X2	8	6	36
3	X3	14	0	0
4	Разом	42	0	72

Для перевірки ступеня достовірності експертних оцінок у методі ранжування застосовують наступні етапи обчислення (ви вже маєте частину таблиці, яка відповідає цим даним):

а) Сума рангів кожного з параметрів і загальна сума рангів:

X1	20

X2	8
X3	14
Разом	42

Ці значення вже присутні у таблиці, яку ви надіслали раніше.

Наступними кроками перевірки достовірності експертних оцінок будуть:

б) Обчислення коефіцієнта узгодженості рангів за формулою Кендала [39]:

$$x^2 = \frac{12*\sum \Delta_i^2}{m^2(n^3-n)} = \frac{12*72}{7^2*(3^3-3)} = \frac{864}{49*24} \approx 0.735 \quad (3.1)$$

Де:

m = 7 – кількість експертів;

n = 3 – кількість параметрів;

$\sum \Delta_i^2$ = 72

в) Критерій узгодженості:

Оскільки χ 2 ≈ 0.735 χ 2 ≈0.735 — значення порівнюють з табличним значенням критерію χ кр 2 χ кр 2 при α = 0.05 α=0.05 і $n - 1 = 2$ n−1=2. З таблиці критичних значень:

$$x^2_{\text{кр}}(0.05; 2) = 5.99 \quad (3.2)$$

Оскільки χ 2 < χ кр 2 χ 2 <χ кр 2 , то узгодженість рангів слабка або оцінки не є статистично достовірними, і необхідно переглянути або уточнити критерії оцінювання [39].

Таблиця 3.5 – Результати ранжування параметрів

Параметр	Ранг параметра	Сума рангів	Відхилення, Δ_i	Δ_i^2

	за оцінкою експерта			
X1	2	3	3	3
X2	1	1	1	1
X3	3	2	2	2
Разом	6	6	6	6

Для розрахунку вагомості кожного параметра (тобто коефіцієнта важливості K_{Bi}на основі числових оцінок переваг, складається матриця попарних порівнянь $A = ||a_{ij}||$, де:

- a_{ij} — числова оцінка переваги параметра iii над параметром jjj;
- $a_{ii} = 1$— діагональні елементи дорівнюють 1, оскільки параметр порівнюється сам із собою;
- якщо $a_{ij} > 1$, то параметр iii більш значимий за jjj;
- якщо $a_{ij} < 1$, то параметр jjj більш значимий за iii;
- $a_{ij} = \frac{1}{a_{ij}}$.

Формула для розрахунку вагомості кожного параметра:

$$K_{Bi} = \frac{\Pi_{j=1}^{n} a_{ij}}{\sum_{i=1}^{n} \Pi_{j=1}^{n} a_{ij}} \quad (3.3)$$

Де:

n – кількість параметрів (у нашому випадку 3: X1, X2, X3),

a_{ij} – значення з матриці попарних порівнянь [40].

Оцінка рівня якості кожного з варіантів реалізації основних функцій програмного продукту проводиться індивідуально на основі абсолютних значень ключових параметрів.

Параметр X1 (об’єм пам’яті, необхідної для обчислень) повністю відповідає технічним вимогам, визначеним умовами експлуатації програмного продукту.

Параметр X2 (час виконання задачі) приймає помірне або оптимальне

значення — тобто обрано не найгірший сценарій. Це відповідає значенням варіантів [41]:

- б) 22 с
- в) 9 с

Параметр X3 (кількість рядків коду, що опосередковано вказує на складність і підтримуваність системи) також не має найгіршого значення.

Враховані два сценарії:

- б) 220 рядків
- в) 130 рядків

Формула розрахунку коефіцієнта технічного рівня якості Для кожного варіанту реалізації програмного продукту коефіцієнт технічного рівня якості $K_{\text{тр}}$ обчислюється за наступною формулою:

$$\text{К}_{\text{тр}} = \sum_{i=1}^{n} * \ K_{Bi} B_i \qquad (3.4)$$

Де:

n – кількість параметрів;

K_{Bi} – коефіцієнт вагомості і–го параметра;

B_i – оцінка i–го параметра в балах.

У таблиці 3.6 наведено результати розрахунку показників якості реалізації програмного продукту. Вона демонструє, як співвідношення параметрів до їх оптимальних значень впливає на загальний коефіцієнт технічного рівня $K\ T$ К Т , що дозволяє обґрунтувати вибір найбільш ефективного варіанта [39-41].

Таблиця 3.6 – Розрахунок Показників Якості

	Варіант	X1	X2	X3

1	А	16	35	500
2	Б	8	22	220
3	В	4	9	130

Визначення рівня якості комбінацій реалізації функцій

На основі попередньо розрахованих коефіцієнтів технічного рівня якості для кожного з параметрів було здійснено підсумкову оцінку чотирьох можливих комбінацій реалізації основних функцій [42]:

- F1(Б) – F2(Б) – F3(Б):
  KT=4,1+4,3+1,6=10,0K_T = 4,1 + 4,3 + 1,6 = 10,0KT=4,1+4,3+1,6=10,0
- F1(Б) – F2(А) – F3(Б):
  KT=4,1+2,15+1,6=7,85K_T = 4,1 + 2,15 + 1,6 = 7,85KT=4,1+2,15+1,6=7,85
- F1(Б) – F2(А) – F3(А):
  KT=4,1+2,15+0,8=7,05K_T = 4,1 + 2,15 + 0,8 = 7,05KT=4,1+2,15+0,8=7,05
- F1(Б) – F2(Б) – F3(А):
  KT=4,1+4,3+0,8=9,2K_T = 4{,}1 + 4{,}3 + 0{,}8 = 9{,}2KT=4,1+4,3+0,8=9,2

3.4 Перспективи розвитку і застосування ігрових CPU-систем

Завдяки своїй високій продуктивності, оптимізації під багатопоточність і здатності працювати в умовах інтенсивного навантаження, ці системи знаходять застосування в різних галузях, що виходять за межі традиційної індустрії відеоігор.

1. Розширення функціональності в межах геймінгу

З розвитком хмарного геймінгу та стрімінгових платформ (як-от GeForce NOW, Xbox Cloud Gaming, Google Stadia) виникає потреба у CPU, здатних паралельно обробляти декілька ігрових сесій, керувати потоками користувачів та забезпечувати мінімальні затримки. Це стимулює розробку процесорів із більшою кількістю ядер, розвиненою кеш-пам’яттю та покращеним енергоспоживанням [43].

2. Застосування у сфері симуляцій та навчання

Ігрові CPU-системи ефективно використовуються в освітніх симуляціях, віртуальній реальності (VR) та доповненій реальності (AR), де важливими є низька затримка й реалістичне моделювання фізичних процесів. Їх також активно інтегрують у тренажери для авіації, медицини, військової підготовки.

3. Використання у дослідницьких і наукових задачах

Багатоядерні ігрові процесори знаходять своє місце в задачах обробки великих масивів даних, edge-computing, а також у сфері глибокого навчання та нейромереж, де застосовуються для попередньої обробки даних до передавання на GPU або сервер.

4. Оптимізація для енергоефективності

Виробники (Intel, AMD) дедалі більше фокусуються на енергоефективності ігрових CPU, що дозволяє використовувати їх у мобільних рішеннях, зменшуючи тепловиділення та споживання електроенергії, не втрачаючи при цьому продуктивності [43].

5. Інтеграція з AI-модулями

Сучасні CPU починають оснащуватись вбудованими модулями для обробки інструкцій штучного інтелекту (AI). Це відкриває нові горизонти у створенні адаптивних ігрових середовищ, автоматичному балансуванні складності, генерації контенту на льоту, а також застосування ігрових рішень у smart-системах.

ВИСНОВКИ

У межах дипломної роботи розглянуто теоретичні засади використання багатоядерних процесорів у паралельних обчисленнях та виокремлено ключові особливості архітектури CPU та GPU, що впливають на вибір технологій реалізації високопродуктивного програмного забезпечення. Проаналізовано переваги багатопотокової обробки даних у порівнянні з послідовною реалізацією, а також підходи до ефективного розподілу обчислювального навантаження між ядрами процесора.

У процесі роботи було розроблено програмний модуль, який демонструє принципи обробки ігрових логічних задач у багатоядерному середовищі. Проєктування системи охоплювало вибір відповідних інструментів та бібліотек для реалізації багатопотоковості (зокрема, Aparapi, TornadoVM), побудову архітектури застосунку, а також організацію взаємодії між потоками.

Продемонстровано методику тестування розробленої системи, що включає вибір критеріїв оцінки, формування сценаріїв навантаження, а також порівняння продуктивності в однопоточному та багатоядерному режимах. Проаналізовано вплив паралелізму на швидкодію, стабільність та масштабованість рішення. Результати експериментів підтверджують доцільність використання Java-орієнтованих фреймворків у задачах інтенсивних обчислень, особливо у гібридних архітектурах із GPU-підтримкою.

На основі проведених розрахунків і побудованих моделей виокремлено варіанти реалізації, які забезпечують найвищий рівень технічної ефективності. Проведено функціонально-вартісний аналіз, що дозволив визначити найкраще співвідношення між ресурсними витратами й досягнутою продуктивністю.

У результаті дослідження сформовано практичні рекомендації щодо впровадження CPU-орієнтованих рішень у сфері обробки ігрових задач, а також визначено перспективи розвитку ігрових CPU-систем у суміжних галузях — від симуляцій і навчального програмного забезпечення до хмарних обчислень і AI-інтегрованих середовищ.

ПЕРЕЛІК ПОСИЛАНЬ

1. Goetz B., Peierls T., Bloch J., Bowbeer J., Holmes D., Lea D. Java Concurrency in Practice. – Boston: Addison-Wesley, 2006. – 384 p.
2. Mattson T.G., He Y., Koniges A.E. The OpenMP Common Core: Making OpenMP Simple Again. – Cambridge, MA: The MIT Press, 2019. – 264 p.
3. Chapman B., Jost G., van der Pas R. Using OpenMP: Portable Shared Memory Parallel Programming. – Cambridge, MA: The MIT Press, 2007. – 384 p.
4. van der Pas R., Stotzer E., Terboven C. Using OpenMP – The Next Step: Affinity, Accelerators, Tasking, and SIMD. – Cambridge, MA: The MIT Press, 2017. – 392 p.
5. Chandra R., Menon R., Dagum L., Kohr D., Maydan D., McDonald J. Parallel Programming in OpenMP. – San Francisco: Morgan Kaufmann, 2000. – 272 p.
6. Quinn M.J. Parallel Programming in C with MPI and OpenMP. – New York: McGraw-Hill, 2003. – 529 p.
7. Bloch J. Effective Java. – 3rd ed. – Boston: Addison-Wesley, 2018. – 416 p.
8. Eckel B. Thinking in Java. – 4th ed. – Upper Saddle River: Prentice Hall, 2006. – 1150 p.
9. Mattson T.G., Sanders B.A., Massingill B.L. Patterns for Parallel Programming. – Boston: Addison-Wesley, 2004. – 496 p.
10. Pacheco P.S. An Introduction to Parallel Programming. – Amsterdam: Morgan Kaufmann, 2011. – 392 p.
11. Oracle Corporation. Java VisualVM Documentation [Електронний ресурс]. – Режим доступу: https://docs.oracle.com/javase/8/docs/technotes/guides/visualvm/
12. OpenJDK. Java Microbenchmark Harness (JMH) [Електронний ресурс]. – Режим доступу: https://openjdk.org/projects/code-tools/jmh/

VisualVM. Getting Started with VisualVM [Електронний ресурс]. – Режим доступу: https://visualvm.github.io/gettingstarted.html

13. OpenMP Architecture Review Board. OpenMP Application Programming Interface Version 5.1 [Електронний ресурс]. – Режим доступу: https://www.openmp.org/wp-content/uploads/OpenMP-API-Specification-5-1.pdf

14. OpenMP. Resources and Documentation [Електронний ресурс]. – Режим доступу: https://www.openmp.org/resources/

15. Träff J.L. Lectures on Parallel Computing [Електронний ресурс] // arXiv. – 2024. – Режим доступу: https://arxiv.org/abs/2407.18795

16. Kadosh T., Hasabnis N., Mattson T.G., Pinter Y., Oren G. Quantifying OpenMP: Statistical Insights into Usage and Adoption [Електронний ресурс] // arXiv. – 2023. – Режим доступу: https://arxiv.org/abs/2308.08002

17. Zhang T., Shirzad S., Diehl P., Tohid R., Wei W., Kaiser H. An Introduction to hpxMP: A Modern OpenMP Implementation Leveraging HPX [Електронний ресурс] // arXiv. – 2019. – Режим доступу: https://arxiv.org/abs/1903.03023

18. Tian S., Chesterfield J., Doerfert J., Chapman B. Experience Report: Writing A Portable GPU Runtime with OpenMP 5.1 [Електронний ресурс] // arXiv. – 2021. – Режим доступу: https://arxiv.org/abs/2106.03219

19. Sutter H. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software [Електронний ресурс] // Dr. Dobb's Journal. – 2005. – Режим доступу: https://www.gotw.ca/publications/concurrency-ddj.htm

20. Baeldung. Microbenchmarking with Java [Електронний ресурс]. – Режим доступу: https://www.baeldung.com/java-microbenchmark-harness

21. Jenkov J. JMH – Java Microbenchmark Harness Tutorial [Електронний ресурс]. – Режим доступу: https://jenkov.com/tutorials/java-performance/jmh.html

22. VisualVM. Official Website [Електронний ресурс]. – Режим доступу: https://visualvm.github.io/

23. OpenMP. Official Website [Електронний ресурс]. – Режим доступу: https://www.openmp.org/

GeeksforGeeks. Java Microbenchmark Harness (JMH) [Електронний ресурс]. – Режим доступу: https://www.geeksforgeeks.org/java-microbenchmark-harness-jmh/

24. Oracle Corporation. Java Platform, Standard Edition 8 API Specification [Електронний ресурс]. – Режим доступу: https://docs.oracle.com/javase/8/docs/api/

25. Oracle Corporation. Java SE Documentation [Електронний ресурс]. – Режим доступу: https://docs.oracle.com/javase/8/docs/

26. OpenJDK. Project Panama: Interconnecting JVM and Native Code [Електронний ресурс]. – Режим доступу: https://openjdk.org/projects/panama/

27. JetBrains. JMH Plugin for IntelliJ IDEA [Електронний ресурс]. – Режим доступу: https://plugins.jetbrains.com/plugin/7529-jmh-java-microbenchmark-harness

28. Stack Overflow. Questions Tagged 'Java Concurrency' [Електронний ресурс]. – Режим доступу: https://stackoverflow.com/questions/tagged/java-concurrency

29. TornadoVM Documentation [Електронний ресурс]. – Режим доступу: https://www.tornadovm.org (дата звернення: 18.05.2025).

30. Aparapi – Native Java for Data-Parallel Programming [Електронний ресурс]. – Режим доступу: https://aparapi.github.io (дата звернення: 18.05.2025).

31. Java ForkJoin Framework [Електронний ресурс]. – Режим доступу: https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html (дата звернення: 18.05.2025).

32. Java Threads Tutorial [Електронний ресурс]. – Режим доступу: https://docs.oracle.com/javase/tutorial/essential/concurrency/ (дата звернення: 18.05.2025).

33. JCuda – Java bindings for CUDA [Електронний ресурс]. – Режим доступу: https://jcuda.org (дата звернення: 18.05.2025).

34. PolyBench Benchmark Suite [Електронний ресурс]. – Режим доступу: https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/ (дата звернення: 18.05.2025).

35. JMH – Java Microbenchmarking Harness [Електронний ресурс]. – Режим доступу: https://openjdk.org/projects/code-tools/jmh/ (дата звернення: 18.05.2025).

CUDA Toolkit Documentation [Електронний ресурс]. – Режим доступу: https://docs.nvidia.com/cuda/index.html (дата звернення: 18.05.2025).

36. Intel Threading Building Blocks (TBB) [Електронний ресурс]. – Режим доступу: https://www.intel.com/content/www/us/en/developer/tools/oneapi/threading-building-blocks.html (дата звернення: 18.05.2025).

37. Java Grande Forum Benchmark Suite [Електронний ресурс]. – Режим доступу: https://www.epcc.ed.ac.uk/research/computing/performance-evaluation-tools/java-grande (дата звернення: 18.05.2025).

38. IBM POWER8 Technical Overview [Електронний ресурс]. – Режим доступу: https://www.ibm.com/docs/en/power8 (дата звернення: 18.05.2025).

39. NVIDIA Tesla K40 GPU Architecture [Електронний ресурс]. – Режим доступу: https://www.nvidia.com/en-us/data-center/tesla-k40/ (дата звернення: 18.05.2025).

40. OpenCL Overview – Khronos Group [Електронний ресурс]. – Режим доступу: https://www.khronos.org/opencl/ (дата звернення: 18.05.2025).

41. Parallel Computing with Java Streams [Електронний ресурс]. – Режим доступу: https://www.baeldung.com/java-8-parallel-streams (дата звернення: 18.05.2025).

42. Multithreaded Game Engine Architecture [Електронний ресурс]. – Режим доступу: https://wickedengine.net/2020/04/26/multithreaded-rendering/ (дата звернення: 18.05.2025).

ДЕМОНСТРАЦІЙНІ МАТЕРІАЛИ

Державний університет інформаційно-комунікаційних технологій

Кафедра комп’ютерної інженерії

Кваліфікаційна робота на тему:

«БАГАТОЯДЕРНІ ПРОЦЕСОРИ У СУЧАСНИХ ІГРОВИХ ДОДАТКАХ»

на здобуття освітнього ступеня бакалавра
зі спеціальності 123 Комп’ютерна інженерія
освітньо-професійної програми Комп’ютерна інженерія

Виконав: Олександр Бандура КІД-41
Науковий керівник роботи: Сергій Коротков

КИЇВ - 2025

Актуальність дослідження. У сучасних умовах стрімкого розвитку ігрової індустрії зростають вимоги до продуктивності програмного забезпечення, особливо в частині обробки складної графіки, фізичних моделей та штучного інтелекту. Традиційні однопоточні підходи вже не забезпечують належного рівня ефективності, що спонукає до активного використання багатоядерних процесорів.

Мета роботи – проаналізувати можливості багатоядерних процесорів у підвищенні продуктивності ігрових застосунків, розробити та протестувати програмний модуль із підтримкою паралельної обробки даних, а також провести оцінку ефективності реалізованих рішень.

Об'єкт дослідження – процеси обробки та оптимізації обчислювальних задач в ігрових додатках із використанням багатоядерних процесорів.

Предмет дослідження – архітектура багатоядерних процесорів, моделі паралельного програмування та програмні засоби реалізації багатопотокових обчислень у сучасних ігрових середовищах.

Паралелізм в обчислювальних системах — це здатність виконувати кілька обчислювальних операцій одночасно, що дає змогу суттєво підвищити швидкодію програм. У сучасних комп'ютерах паралелізм реалізується на різних рівнях: на рівні інструкцій (ILP), потоків (TLP) і процесів (PLP). Кожен із цих підходів дозволяє ефективно використовувати апаратні ресурси, зокрема багатоядерні процесори.

Для ігрових застосунків паралелізм є критично важливим, оскільки дозволяє обробляти графіку, фізику, логіку гри та штучний інтелект одночасно, без затримок. В умовах постійного зростання вимог до якості зображення та складності сцен розподіл навантаження між ядрами CPU стає ключовим фактором у забезпеченні високої продуктивності.

Архітектура багатоядерних процесорів: CPU vs GPU



Рисунок 1.1 – Central Processing Unit



Рисунок 1.2 – Графічний процесор (GPU)

Таблиця 1.1 – Порівняльна характеристика GPU та GPU

Критерій	CPU	GPU
Кількість ядер	4–32	1000+ (малі ядра)
Тип обчислень	Послідовні/розгалужені	Масово-паралельні (SIMD)
Гнучкість логіки	Висока	Низька (умовні розгалуження дорогоцінні)
Пропускна здатність	Середня	Висока
Затримка доступу до пам’яті	Низька	Вища
Споживання енергії	Помірне	Високе при повному навантаженні
Застосування в іграх	Ігрова логіка, ШІ, ІО	Рендеринг, фізика, обробка зображень

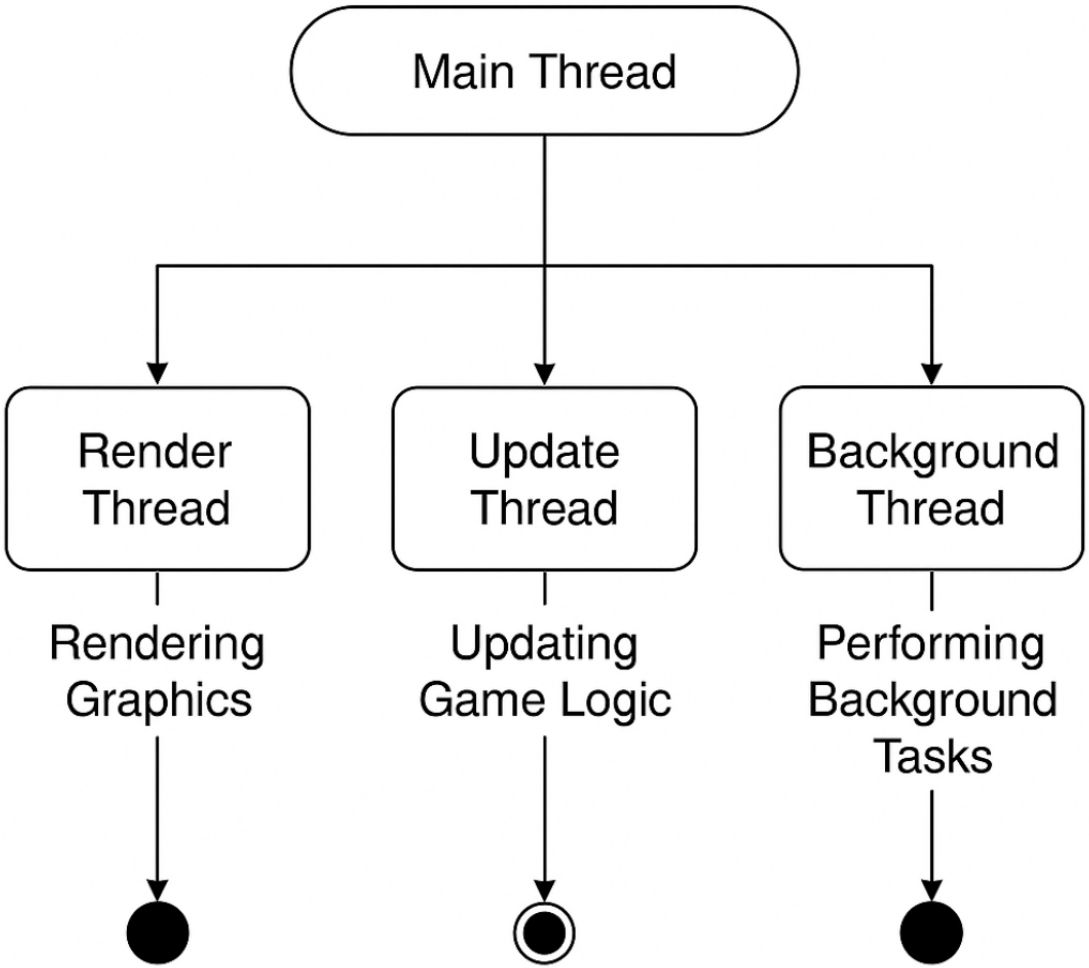


Рисунок 1.3 – UML-діаграма потокової архітектури Java-гейменджина: основні потоки обробки

Таблиця 1.2 – Порівняльна характеристика Aparapi та TornadoVM для GPU-обчислень у Java

Критерій	Aparapi	TornadoVM
Рік появи	~2011	~2019
Підхід до обчислень	Java → OpenCL	Java → GraalVM → OpenCL / PTX / SPIR-V
Підтримка GPU	Через OpenCL	Через OpenCL, PTX (NVIDIA), SPIR-V (Intel, AMD, Vulkan)
Мови програмування	Тільки Java	Java, Kotlin, Scala
Автоматична компіляція	Так (оголошені методи транслюються у OpenCL)	Так (автоматичне визначення обчислювальних графів)
Можливість fallback на CPU	Так	Так
Підтримка багатопристроєвих систем	Обмежена	Повна (CPU, GPU, FPGA одночасно)
Гнучкість налаштувань та оптимізацій	Середня	Висока (аналіз графу задач, профілювання під час виконання)
Сумісність із екосистемою Java	Висока	Висока (але потребує GraalVM)
Складність інтеграції	Низька (простий API)	Середня (потрібна конфігурація GraalVM, компілятора TornadoVM)
Сфера застосування	Просте GPU-прискорення (частки, NPC-логіка)	Складні гетерогенні обчислення (рендеринг, трасування променів, AI, тощо)

Оптимізація навантаження на ядра процесора

- Раціональне використання багатоядерного процесора є основою стабільної роботи ігрового застосунку. Завдяки правильному розподілу задач між ядрами вдається зменшити затримки та уникнути конфліктів при виконанні паралельних обчислень. У межах реалізованої архітектури потоки розділені за функціональними напрямами: фізика, графіка, AI тощо. Такий підхід покращує кешування і мінімізує синхронізацію.
- Для виявлення вузьких місць використано Java VisualVM і JMH. Було виявлено перевантаження фізичного модуля, що дало змогу виділити окремий пул і оптимізувати обчислення. У результаті були зменшені критичні секції, покращено розподіл потоків і забезпечено стабільні показники FPS.

```
@Benchmark
@BenchmarkMode(Mode.AverageTime)
@OutputTimeUnit(TimeUnit.MILLISECONDS)
public void testPhysicsUpdate() {
    physicsModule.update(0.016f);
}
```

Рисунок 1.4 – Приклад використання JMH для бенчмарку методу оновлення фізики у грі

Порівняльний аналіз продуктивності (багатоядерність vs однопоточність)


```
1.class Compute {
2.    public static void matrixMultiplication(final float[] A,
   final float[] B, final float[] C, final int size) {
3.        for (@Parallel int i = 0; i < size; i++) {
4.            for (@Parallel int j = 0; j < size; j++) {
5.                float sum = 0.0f;
6.                for (int k = 0; k < size; k++)
7.                    sum += A[(i * size) + k] * B[(k * size) + j];
8.                C[(i * size) + j] = sum;
9.            }
10.         }
11.    }
12.}
```

Рисунок 1.5 - Приклад реалізації множення матриць на Java

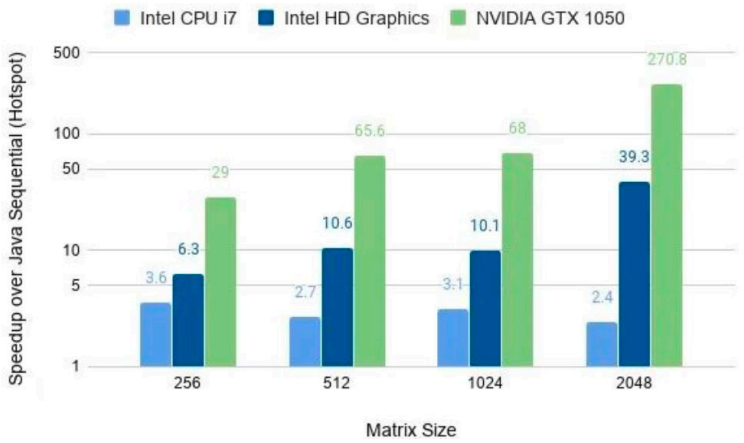


Рисунок 1.6 – Порівняльна швидкодія TornadoVM на різних пристроях при множенні матриць різного розміру

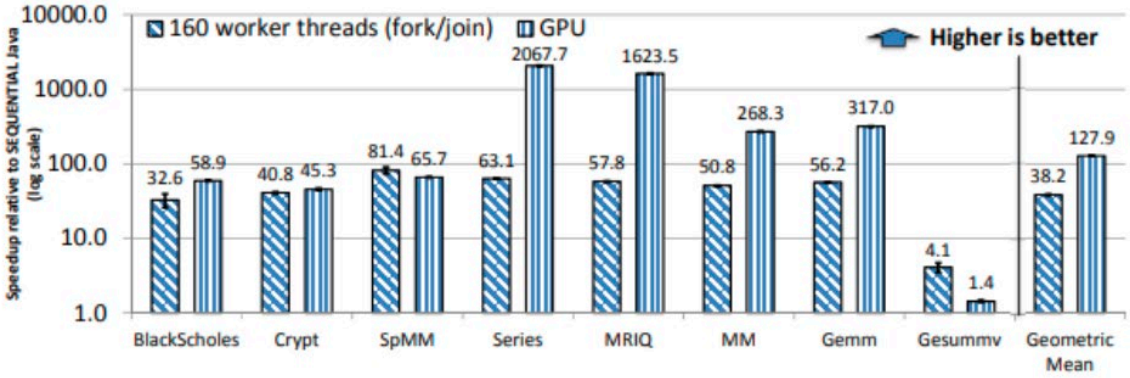


Рисунок 1.7 – Прискорення виконання Java-коду на GPU у порівнянні з CPU

Перспективи розвитку і застосування ігрових CPU-систем

Сучасні ігрові CPU-системи демонструють високу продуктивність і здатність до масштабованого паралельного виконання, що дозволяє їх застосовувати не лише у геймінгу. Вони активно використовуються у хмарному геймінгу, де обробляють багато сесій одночасно з мінімальними затримками. У VR/AR симуляціях такі процесори забезпечують реалістичну фізику та стабільність.

У науці й аналітиці багатоядерні CPU ефективно виконують попередню обробку даних для AI. Окрему увагу приділено енергоефективності — нові чипи працюють з меншими витратами енергії, що актуально для мобільних пристроїв. З інтеграцією AI-модулів CPU отримують нові функції, як-от генерація контенту та інтелектуальне керування ігровим середовищем.

ВИСНОВКИ

- Розглянуто теоретичні засади використання багатоядерних процесорів для паралельних обчислень, проаналізовано відмінності між архітектурами CPU та GPU та їх вплив на продуктивність програмних систем.

- Розроблено програмний модуль, орієнтований на багатопотокову обробку ігрових задач із використанням Java-технологій, зокрема Aparapi та TornadoVM, з урахуванням ефективного розподілу навантаження.

- Продемонстровано методику тестування системи, проведено порівняльний аналіз продуктивності у послідовному та паралельному режимах, що підтвердило переваги багатоядерності.

- Визначено перспективи розвитку ігрових CPU-систем у галузях VR/AR, симуляцій, хмарних обчислень та інтелектуальних середовищ, що підтверджує актуальність і практичну цінність проведеного дослідження.