Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

Курсова робота

на тему:

“Реалізація генетичних алгоритмів ранжування текстової колекції документів з використанням архітектури Nvidia CUDA”

Керівник курсової роботи
доктор технічних наук, доцент
Глибовець А. М.
(прізвище та ініціали)
_____
(підпис)
«____»_____2020 р.
Виконав студент 1 курсу МП
Семилітко М.Ю.
(прізвище та ініціали)
«____»_____2020 р.

Київ – 2020

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри інформатики,

к.ф.-м.н., доц.

_____ Гороховський С.С.

(підпис)

„____”_____2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ

на курсову роботу

студенту Семилітку М.Ю. факультету інформатики 5 курсу

ТЕМА «Реалізація генетичних алгоритмів ранжування текстової колекції документів з використанням архітектури Nvidia CUDA»

Вихідні дані:

- Програмні засоби ранжування текстової колекції документів з використанням архітектури Nvidia CUDA
- Текстова частина курсової роботи

Зміст ТЧ до курсової роботи:

Індивідуальне завдання
1. Вступ
2. Опис алгоритмів
3. Реалізація алгоритму
4. Тестування
5. Висновки

Список використаної літератури
Додатки

Дата видачі “___” _____ 2020 р. Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Календарний план виконання курсової роботи

Тема: «Реалізація генетичних алгоритмів ранжування текстової колекції документів з використанням архітектури Nvidia Cuda»

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу.	06.12.2019	
2.	Огляд технічної літератури за темою роботи.	19.12.2019	
3.	Аналіз алгоритмів ранжування.	31.01.2020	
3.	Аналіз еволюційних алгоритмів.	14.02.2020	
4.	Огляд архітектури Nvidia Cuda.	01.03.2020	
6.	Розробка алгоритму.	07.04.2020	
7.	Написання пояснювальної роботи.	20.04.2020	
8.	Створення слайдів для доповіді та написання доповіді.	27.04.2020	
8.	Аналіз отриманих результатів з керівником.	07.05.2020	
11.	Остаточне оформлення пояснювальної роботи та слайдів.	09.05.2020	
12.	Захист курсової роботи		

Студент: **Семилітко М.Ю.**

Керівник: **Глибовець А.М.**

“_____”_____

Анотація

В курсовій роботі розроблюється паралельний еволюційний алгоритм з використанням архітектури Nvidia CUDA для ранжування текстової колекції документів. Для цього в роботі проводиться аналіз існуючих алгоритмів ранжування, які використовуються в пошукових системах, розглядаються різні варіанти еволюційних алгоритмів, а також розглядаються основні аспекти роботи програм, які виконуються обчислення на відеокартах. На основі цього, було створене програмне забезпечення, яке дозволяє визначати ваги для різних факторів ранжування.

Перший розділ розглядає базовий алгоритм ранжування пошукової системи Google, а також виділяє основні фактори, які використовуються в системах пошуку для сортування веб-сторінок за корисністю для користувача. Далі розглядаються різні версії еволюційних алгоритмів за методами роботи з популяцією. Після цього описується взаємодія програм з архітектурою CUDA.

Другий розділ присвячений детальному опису розробки та роботи паралельного алгоритму, за допомогою якого можна визначати ваги для факторів ранжування веб-сторінок.

В третьому розділі проводиться тестування розробленого алгоритму на швидкість роботи за різних розмірів популяції, розмірів генів та тестових наборів даних.

Ключові слова: еволюційний алгоритм, генетичний алгоритм, паралельний генетичний алгоритм, ранжування текстової колекції документів, Nvidia CUDA.

Зміст

1. Вступ

З кожним днем кількість інформації в інтернеті нестримно збільшується, з'являється все більше і більше джерел інформації, які можуть бути як корисними так і ні. Це стає справжнім викликом для систем інформаційного пошуку, правильно ранжувати сторінки за запитом користувача, тому пошукові системи додають нові алгоритми та фактори, за якими обчислюється ранг веб-сторінки. Уже цього року, кількість усіх компонентів для оцінки сторінок у компанії Google становить більше 200.

Зі збільшенням кількості даних, зростає й час їх обробки, тому необхідно, щоб алгоритм опрацювання був швидким, але міг забезпечити гарні результати. Якщо розглянути усі фактори як невідомі змінні, а ранг сторінки як невідоме число, отримаємо функцію багатьох змінних. Для розв'язку такої задачі можуть бути застосовані еволюційні алгоритми. Не можна пропустити той факт, що дані алгоритми мають багато модифікацій, які дозволять отримати приріст швидкості обчислювань різними методами, одним з яких, є використання обчислювальних потужностей графічних чіпів (відеокарт), наприклад архітектури Nvidia CUDA. Її основною перевагою є можливість виконувати велику кількість обчислювань паралельно, що робить її надзвичайно популярною на сьогоднішній день.

В даній роботі буде представлений опис та реалізація алгоритму еволюції з використанням архітектури CUDA, який дозволить проводити ранжування веб-ресурсів за великою кількістю критеріїв, а саме - це обчислення ваг для факторів оцінювання.

2. Опис алгоритмів

2.1 Алгоритми ранжування

Ранжування - це сортування джерел інформації (веб-сайтів), яке використовується пошуковими системами. Алгоритми ранжування постійно оновлюються, оскільки сайти використовують методи сірої та чорної оптимізації.

Спочатку пошукові системи спирались тільки на ранжування за схожістю запиту користувача та контенту ресурсу, що працювало не завжди добре.

В своїй роботі Сергій Брін та Ларрі Пейдж зазначають, що повнота індексу - не єдиний показник якості результатів пошуку [1].

Тому мета алгоритму PageRank - визначити відносний рейтинг сторінки.

Ідея алгоритму базується на алгоритмі оцінки наукових робіт, тобто спираючись на кількість джерел, які посилаються на певну сторінку (Рисунок 2.1.1). [2]

Рисунок 2.1.1 Приклад роботи алгоритму PageRank

В алгоритмі PageRank враховуються такі фактори:

- Кількість та якість вхідних посилань
- Кількість вихідних посилань на кожній сторінці
- Рейтинг кожної сторінки

[2]

В формулі PageRank присутній коефіцієнт затухання, який симулює ймовірність того, що користувач буде переходити за посиланням на сторінках.

Сьогодні основними критеріями ранжування є:

- Посилання на домен
- Кількість переходів за посиланням
- Авторитет домену
- Якість мобільної версії
- Час завантаження сторінки
- Загальна кількість зворотних посилань
- Якість вмісту
- Оптимізація контенту сторінки

[3]

2.2 Еволюційні алгоритми

Концепцію еволюційних алгоритмів запровадив Лоуренс Фогель в 1965 році.

Еволюційні алгоритми - це адаптивний пошуковий механізм, в якому моделюється процес біологічної еволюції. Еволюційні алгоритми можна поділити на генетичні алгоритми, генетичне програмування,

еволюційне програмування, еволюційні стратегії, в яких моделюються такі етапи:

- Відбір
- Схрещення
- Мутація
- Селекція

Кожен з групи еволюційних алгоритмів фокусується на певній особливості природньої еволюції, наприклад, еволюційне програмування зосереджується на адаптації популяції, тоді як генетичні алгоритми використовують схрещення в пошуку, а еволюційні стратегії зосереджені на мутації.

Еволюційні алгоритми застосовуються при класифікації об'єктів, кластеризації документів, локальному пошуку, машинному перекладі, розпізнаванні мовлення та рукописного введення.

Рішеннями задачі виступають особи популяції. Якість рішення оцінюється спеціальною фітнес-функцією, яка задається для кожної задачі окремо, після чого відбувається еволюція популяції.

1. Генерація початкової популяції
2. Оцінка популяції фітнес-функцією
3. Вибираються найкраще пристосовані особини для розмноження
4. Схрещення відібраних особин
5. Мутація генів
6. Оцінка популяції
7. Селекція

До переваг генетичних алгоритмів можна віднести:

1. Можна застосувати до великої кількості задач оптимізації

2. Легко комбінуються з іншими методами
3. Велика кількість одночасних рішень задачі

Недоліками є:

1. Не гарантується знаходження найкращого рішення задачі
2. Слабка масштабованість

2.3 Технологія Nvidia CUDA

Програмно-апаратна технологія CUDA була представлена в 2006 році американською компанією Nvidia. З появою складних алгоритмів, які використовують велику кількість даних, дана технологія зарекомендувала себе, оскільки вона дозволяє виконувати велику кількість обчислень паралельно, що дозволяє їй бути швидшою ніж процесор.

В архітектурі CUDA є 3 основні абстракції:

1. Ієрархія блоків потоків
2. Спільна пам’ять
3. Синхронізація з використанням бар’єрів

[4]

Архітектура CUDA дозволяє розділити алгоритм на великі задачі, кожен з яких складається з малих підзадач. Різниця між задачею та підзадачею полягає в частоті обміну даними між потоками, які виконують обчислення. Тоді кожна задача виконується в окремому блоці потоків, а кожна підзадача в окремому потоці блоку.

Блок потоків - абстракція, яка представляє групу потоків зі спільною пам’яттю (Рисунок 2.1), тому кількість потоків в блоці залежить від розміру спільної пам’яті. Для останнього покоління відеокарт Turing кількість потоків в блоці 512 або 1024. Всі потоки в блоці можуть

обмінюватись даними через спільну пам’ять, використовуючи атомарні змінні або бар’єри.

Об'єднання блоків потоків створює сітку. Всі блоки в сітці мають однакову кількість потоків та розмір спільної пам’яті. Сітки створюються для того, щоб велика кількість блоків потоків працювали паралельно, що дозволяє збільшити загальну кількість потоків.

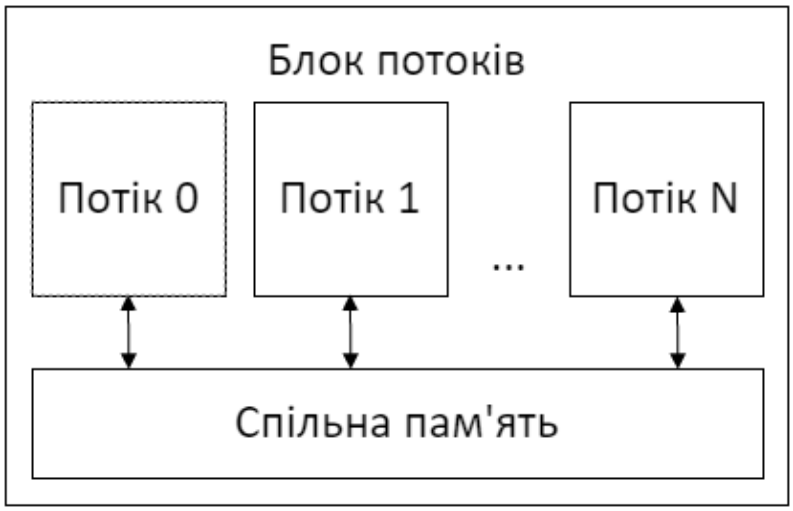

Рисунок 2.1 Приклад блоку потоків

Оскільки програми можуть використовувати багатовимірні масиви даних, блоки потоків та сітки блоків можна організовувати як 1D (Рисунок 2.2), 2D (Рисунок 2.3) та 3D (Рисунок 2.4) масиви. Обмін даними між блоками в сітці неможливий, тому кожна функція блоку повинна працювати самостійно.

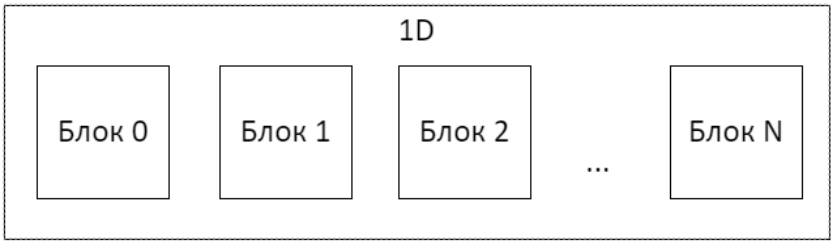

Рисунок 2.2 1D-індексація блоків потоків

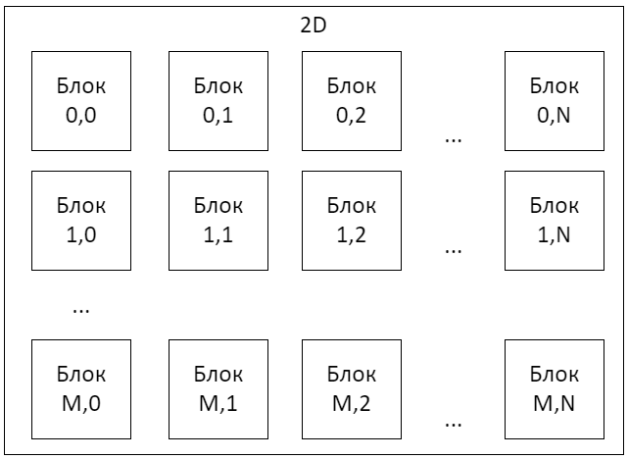

Рисунок 2.3 2D-індексація блоків потоків

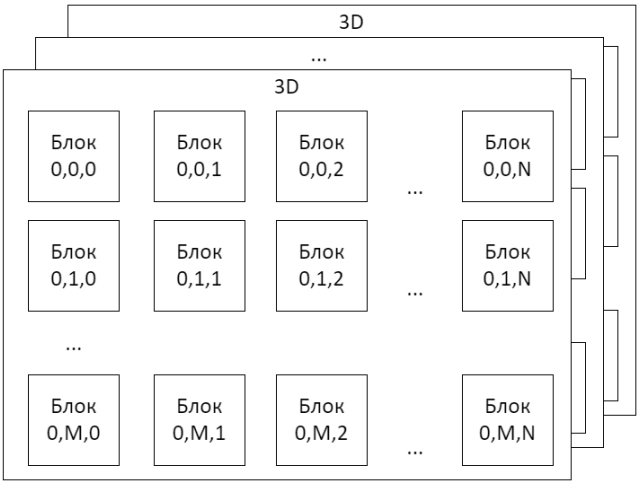

Рисунок 2.4 3D-індексація блоків потоків

Потік - абстракція, яка представляє виконання ядра. Ядро - програма або функція. Кожен потік має індекс, який використовується для обчислення адрес в пам’яті, а також свою локальну пам’ять.

При обробці даних, які представлені у вигляді масиву, зазвичай, для кожного елементу виділяється окремий потік. Для цього обчислюється індекс за формолую (2.1) [5]

$$\mathrm{ID} = \mathrm{cuda.threadIdx.x} + \mathrm{cuda.blockIdx.x} * \mathrm{cuda.blockDim.x} \quad (2.1)$$

Всі необхідні дані зберігаються в глобальній пам’яті. В ній процесор виділяє та очищує пам’ять процесором, а також завантажує необхідні дані.

Доступ до глобальної пам’яті має процесор і всі потоки в блоках (Рисунок 2.5).

Кешовані дані та текстури зберігаються в спеціальних місцях пам’яті з правом доступу тільки на читання. Основне призначення даної пам’яті зберігання текстур, тому ця пам’ять більше оптимізована під операції роботи з тесктурами.

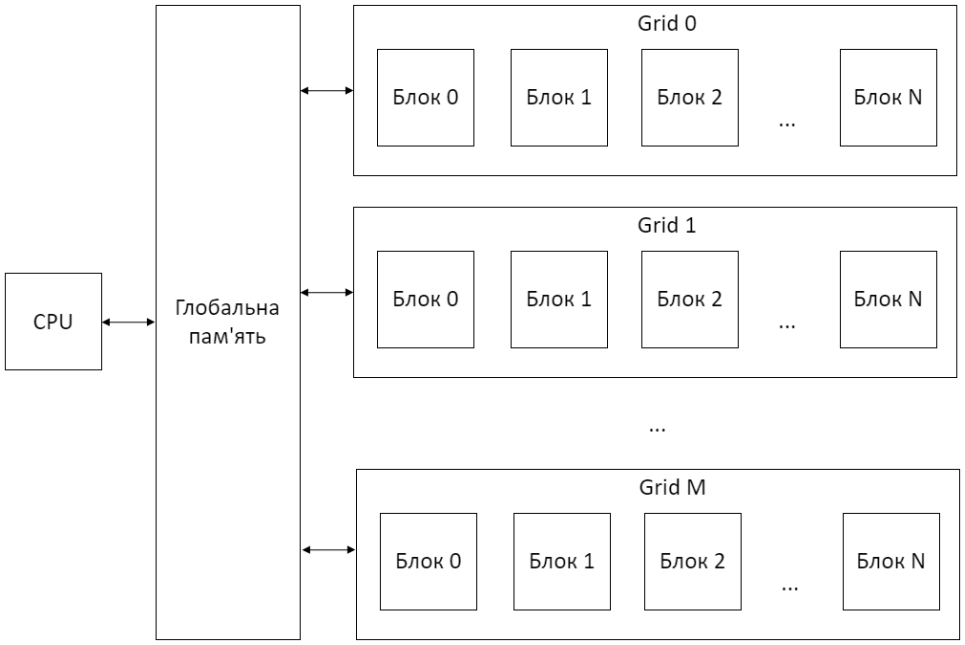

Рисунок 2.5 Організація глобальної пам’яті

3. Реалізація алгоритму

Алгоритм складається з наступних етапів:

1. Ініціалізація даних для оцінки популяції
2. Виділення необхідної пам’яті
3. Генерація популяції
4. Оцінка популяції фітнес-функцією
5. Турнірний відбір найкращих особин з найкращими генами
6. Схрещення відібраних екземплярів
7. Мутація нащадків
8. Оцінка нового покоління фітнес-функцією
9. Селекція

Пункти 5 – 9 можуть повторюватись.

Популяція складається з множини генів, які в процесі еволюції будуть відбиратись, схрещуватись та мутувати за певними принципами. Кожен ген складається з хромосом. Кожна хромосома - це ваговий коефіцієнт для фактору ранжування, представлений у вигляді дійсного числа. Кількість факторів визначає розмір генів, тобто кількість хромосом.

Для того, щоб збільшити швидкість роботи алгоритму, вся необхідна пам’ять виділяється завчасно, а всі дані зберігаються у вигляді одновимірних масивів, що підвищує рейтинг влучень по кешу. [6]

Представлений алгоритм немає жорсткої прив’язки до кількості хромосом в гені, тому його не потрібно перероблювати, якщо кількість факторів змінилась.

Для розробки алгоритму було використано мову програмування C++11, CUDA v10.1, компілятор nvcc v10.1.105, середовище розробки Visual Studio 2017, SDK version 10.0.18362.0.

3.1 Структура вхідних даних

Вхідні дані представлені у вигляді структури, яка включає в себе:

- Матриця дійсних чисел, у якому кожне число строки відповідає оцінці джерела інформації у відповідності до певного фактору, наприклад косинусна міра схожості, ранг домену, якість вмісту, якість мобільної версії, тощо, а кількість строк залежить від кількості ресурсів для ранжування.
- Відсортовані за спаданням перегляду сторінки.

```
struct Dataset_Record
{
   std::list<std::vector<double>> ranks_for_factors;
   std::list<size_t> expected_answer;
};
```

Така структура дозволяє відтворити процес ранжування та забезпечити зручну оцінку популяції.

3.2 Генерація популяції

Для того, щоб підвищити швидкість роботи алгоритму, вся популяція буде представлена як стрічка, в якій зберігаються всі гени. Оскільки всі гени мають фіксований розмір, ми можемо звертатись до будь якого через зміщення в пам’яті. (3.2.1)

$$A[i][j] = A[i * size + j] \quad (3.2.1)$$

Популяція генерується паралельно на відеокарті, оскільки кожен ген може бути опрацьований окремо, після чого з ним можна працювати. Наприклад, популяція з 8192 особин створювалась в 16 блоках, в кожному з яких було задіяно 512 потоків.

Функція для генерування популяції приймає вказівник на всю популяцію, де потім кожний потік візьме свій елемент через зміщення в пам’яті в залежності від свого ID, розмір гену та параметри необхідні для псевдо-випадкового генерування значень хромосом.

Розмір кожного гену - це кількість факторів ранжування плюс їх пристосованість. Це було зроблено для того, щоб збільшити рейтинг кешу, оскільки кожен потік працює саме з геном, тому в програмі для обчислення зміщень використовується саме розмір гену + 1.

Необхідно звернути увагу на те, що функції, які виконуються на відеокарті, не можуть використовувати функції хоста, такі як rand() або malloc(), тому тут використовується curandState, для генерації випадкових чисел в середені блоків потоків.

```
__global__ void generate_population(double* population, const size_t chr_size, const double min_v, const double max_v, curandState* state)
  {
    size_t thread_id = threadIdx.x + blockDim.x*blockIdx.x;
    size_t block_id = thread_id * (chr_size + 1);
    for (size_t i = 0; i < chr_size; ++i)
      population[block_id + i] = (max_v - min_v) * rand_double_on_device(thread_id, state);
 }
```

3.3 Фітнес функція

Обчислення коефіцієнта придатності відбувається для всіх генів паралельно, оскільки пристосованість одного не залежить від інших. Для всіх вхідних тестових наборів спочатку підраховуємо загальний рейтинг

кожного джерела, потім сортуємо їх за спаданням і порівнюємо з списком з набору. В цьому випадку пристосованість - це відношення кількості однакових значень до всіх.

Оскільки можуть бути тестові дані з різною кількістю сторінок для ранжування, завчасно виділяється пам’ять під розрахунки (double* sum - пам’ять для обчислення сум, unsigned long* - пам’ять для створення списків), де кожен потік може використовувати певну частину цієї пам’яті (обмежено max_training_size * chr_size).

```
__device__ double calc_fitness(double* chromosome,
    const double* koef,
    const unsigned long* expected_result,
    const unsigned long* sizes,
    const size_t chr_size,
    const size_t dataset_size,
    const size_t max_training_size,
    double* sum,
    unsigned long* chr_result,
    const size_t id,
    const size_t shared_memory_offset){
    unsigned long count = 0;
    unsigned long total = 0;
    unsigned long test_id = 0;
```

```
for (size_t i = 0; i < dataset_size; ++i) {
    for (size_t j = 0; j < sizes[i]; ++j) {
        double tmp = 0.0;
        for (size_t k = 0; k < chr_size; ++k)
            tmp += chromosome[id + k] * koef[total * chr_size + k];
        sum[shared_memory_offset + j] = tmp;
        chr_result[shared_memory_offset + j] = j;
        ++total;
    }
    double tmp;
    bool swapped = false;
    do {
        swapped = false;
        for (size_t j = 1; j < sizes[i]; ++j)
            if (sum[shared_memory_offset + j] >
                    sum[shared_memory_offset + j - 1]) {
                swapped = true;
                tmp = sum[shared_memory_offset + j];
                sum[shared_memory_offset + j] =
                    sum[shared_memory_offset + j - 1];
```

```
                sum[shared_memory_offset + j - 1] = tmp;
                unsigned long long tmp_id =
                        chr_result[shared_memory_offset + j];
                chr_result[shared_memory_offset + j] =
                        chr_result[shared_memory_offset + j - 1];
                chr_result[shared_memory_offset + j - 1] = tmp_id;
            }
        } while (swapped);
        for (size_t j = 0; j < sizes[i]; ++j)
            if (chr_result[shared_memory_offset + j] == expected_result[test_id +
j])
                count += 1;
        test_id += sizes[i];
    }
    return (double)count / (double)total;
}
```

3.4 Відбір популяції

Відбір з популяції відбувається за турнірним принципом. З всієї популяції вибирається певна кількість гравців, потім серед них вибирається найкращий. Найчастіше проводять парні турніри, тобто вибирають по 2 гена з популяції. [7]

Перевагою даного алгоритму відбору є відсутність додаткових обчислень та сортувань, а також його легко виконувати паралельно.

```
__global__ void tournament(const double* chromosome,
    const size_t chr_size,
    const size_t population_size,
    const size_t players_cnt,
    double* selected,
    curandState * state)
{
  size_t stateID = blockDim.x * blockIdx.x + threadIdx.x;
  size_t id = (blockDim.x * blockIdx.x + threadIdx.x) * (chr_size + 1);
  size_t best_id = rand_on_device(stateID, state) % population_size;
  for (size_t i = 0; i < players_cnt - 1; ++i)
  {
    size_t random_ul = rand_on_device(stateID, state) % population_size;
    if (chromosome[best_id * (chr_size + 1) + chr_size] < chromosome[random_ul * (chr_size + 1) + chr_size])
      best_id = random_ul;
  }
  best_id *= chr_size + 1;
  for (size_t i = 0; i <= chr_size; ++i)
```

```
    selected[id + i] = chromosome[best_id + i];
}
```

3.5 Схрещення та мутація

Для схрещення було обрано лінійний кросовер. Створюється 3 нащадки, які розраховуються за формулами (3.5.1) - (3.5.3), з яких вибирається 2 найбільш пристосованих. [8]

$$child_i^1 = \left(parent_i^1 + parent_i^2\right) / 2 \quad (3.5.1)$$

$$child_i^2 = \left(3 * parent_i^1 - parent_i^2\right) / 2 \quad (3.5.2)$$

$$child_i^3 = \left(3 * parent_i^2 - parent_i^1\right) / 2 \quad (3.5.3)$$

Оскільки схрещення виконується паралельно на відеокарті, необхідно створити буфер великого розміру, куди потоки зможуть записувати створені гени та обчислювати їх придатність.

```
__global__ void crossover(const double* population,
    const size_t chr_size,
    const size_t* sorted_ids,
    const size_t popultaion_size,
    double* new_popultaion,
    const double* koef,
    const unsigned long* expected_result,
    const unsigned long* sizes,
    const size_t dataset_size,
    const size_t max_training_size,
```

```
    double * sum,
    unsigned long * chr_result,
    double * shared_memory )
{
    size_t thread_id = blockDim.x * blockIdx.x + threadIdx.x;
    size_t id = thread_id * (chr_size + 1);
    size_t shared_memory_id = thread_id * 3 * (chr_size + 1);
    size_t shared_memory_offset = thread_id * max_training_size;
    size_t father = sorted_ids[thread_id / 128] * (chr_size + 1);
    size_t mother = sorted_ids[thread_id % 128 + 1 + thread_id / 128] *
(chr_size + 1);
    size_t best[2];
    for (size_t i = 0; i < chr_size; ++i)
    {
        double tmp;
        tmp = (population[father + i] + population[mother + i]) / 2.0;
        if (tmp < 0.0)
            tmp *= -1.0;
        shared_memory[shared_memory_id + i] = tmp;
        tmp = (3 * population[mother + i] - population[father + i]) / 2.0;
        if (tmp < 0.0)
```

```
            tmp *= -1.0;
        shared_memory[shared_memory_id + i + (chr_size + 1)] = tmp;
        tmp = (3 * population[father + i] - population[mother + i]) / 2.0;
        if (tmp < 0.0)
            tmp *= -1.0;
        shared_memory[shared_memory_id + i + (chr_size + 1) * 2] = tmp;
    }
    double fitness_1 = calc_fitness(shared_memory, koef, expected_result, sizes, chr_size, dataset_size, max_training_size, sum, chr_result, shared_memory_id, shared_memory_offset);
    double fitness_2 = calc_fitness(shared_memory, koef, expected_result, sizes, chr_size, dataset_size, max_training_size, sum, chr_result, shared_memory_id + (chr_size + 1), shared_memory_offset);
    double fitness_3 = calc_fitness(shared_memory, koef, expected_result, sizes, chr_size, dataset_size, max_training_size, sum, chr_result, shared_memory_id + (chr_size + 1) * 2, shared_memory_offset);
    shared_memory[shared_memory_id + chr_size] = fitness_1;
    shared_memory[shared_memory_id + 2 * chr_size + 1] = fitness_2;
    shared_memory[shared_memory_id + 3 * chr_size + 2] = fitness_3;
    if (fitness_1 < fitness_2){ best[0] = 1;
        if (fitness_1 < fitness_3) best[1] = 2; else best[1] = 0; }
    else{ best[0] = 0;
```

```
        if (fitness_2 < fitness_3) best[1] = 2; else best[1] = 1;}

      for (size_t i = 0; i < 2; ++i){ size_t new_popultaion_offset = id * 2 + i * (chr_size + 1);

        size_t shared_memory_copy_offset = shared_memory_id + best[i] * (chr_size + 1);

        for (size_t j = 0; j < chr_size; ++j)

          new_popultaion[new_popultaion_offset + j] = shared_memory[shared_memory_copy_offset + j];

        new_popultaion[new_popultaion_offset + chr_size] = shared_memory[shared_memory_copy_offset + chr_size];

      }

   }
```

Далі кожен ген випадково може мутувати, тобто може змінити значення однієї хромосоми в межах норми.

```
__global__ void mutation(double* population, const size_t chr_size, curandState * state, const double min_v, const double max_v){

    size_t stateID = blockDim.x * blockIdx.x + threadIdx.x;

    size_t id = (blockDim.x * blockIdx.x + threadIdx.x) * (chr_size + 1);

    if (rand_on_device(stateID, state) % 2){

        size_t rand_chromosome = rand_on_device(stateID, state) % chr_size;

        population[id + rand_chromosome] = rand_double_on_device(stateID, state) * (max_v - min_v);

    }
```

```
}
```

3.6 Оцінка та заміщення популяції

Оцінка популяції виконується в 2 етапи. На першому, гени розбиваються на сегменти, де 1 сегмент оброблює 1 потік відеокарти, а саме обчислює середню пристосованість та найкращий ген сегменту. __global__ void accuracy(const double* population, const size_t chr_size, const size_t count, const size_t block_size, double* results, size_t* max_acc_id) {

```
    size_t thread_id = blockDim.x * blockIdx.x + threadIdx.x;

    size_t id = (blockDim.x * blockIdx.x + threadIdx.x) * (chr_size + 1) * block_size;

    double block_sum = 0.0;

    double max_acc = 0.0;

    size_t max_id = 0;

    for (size_t i = 0; i < block_size; ++i){

        size_t p_id = id + (chr_size + 1) * i + chr_size;

        block_sum += population[p_id];

        if (population[id + chr_size] > max_acc){

            max_acc = population[id + chr_size];

            max_id = id + (chr_size + 1) * i;

        }

    }

    results[thread_id] = block_sum / block_size;
```

```
    max_acc_id[thread_id] = max_id;
}
```

Далі інформація копіюється в оперативну пам’ять, де процесор оцінює загальну середню пристосованість популяції, зберігає та оцінює найкращий ген. Якщо його придатність більше заданої, алгоритм припиняє свою роботу, якщо ж ні, порівнюються стара та нові популяції, якщо нова краще, вона повністю заміщує стару, для того щоб уникнути локальних екстремумів, і алгоритм продовжує свою роботу.

Після того як алгоритм завершує свою роботу, отримані ваги факторів можна використовувати для ранжування в пошукових системах.

Основні переваги даного алгоритму:

- Велика кількість одночасних обчислень за допомогою архітектури NVIDIA CUDA
- Мала кількість алокацій пам’яті та копіювань даних
- Кеш-оптимальність
- Гнучкість щодо кількісті розміру генів та вхідних даних

Недоліки:

- Алгоритм не забезпечує найкращий розв’язок
- Точність залежить від кількості та якості вхідних даних
- Складна та повільна фітнес-функція
- Операції на процесорі
- Використовується покращене сортування бульбашкою

4. Тестування

Алгоритм буде тестуватись на швидкість з різним розміром популяції, гену та тестових даних. Всі тестові дані генеруються випадково.

Зі збільшенням розмірів популяції, час 1 ітерації збільшується приблизно в 2^n (Рисунок 4.1), тому що в алгоритмі використовується сортування на процесорі, у якого кількість ітерацій теж приблизно 2^n.

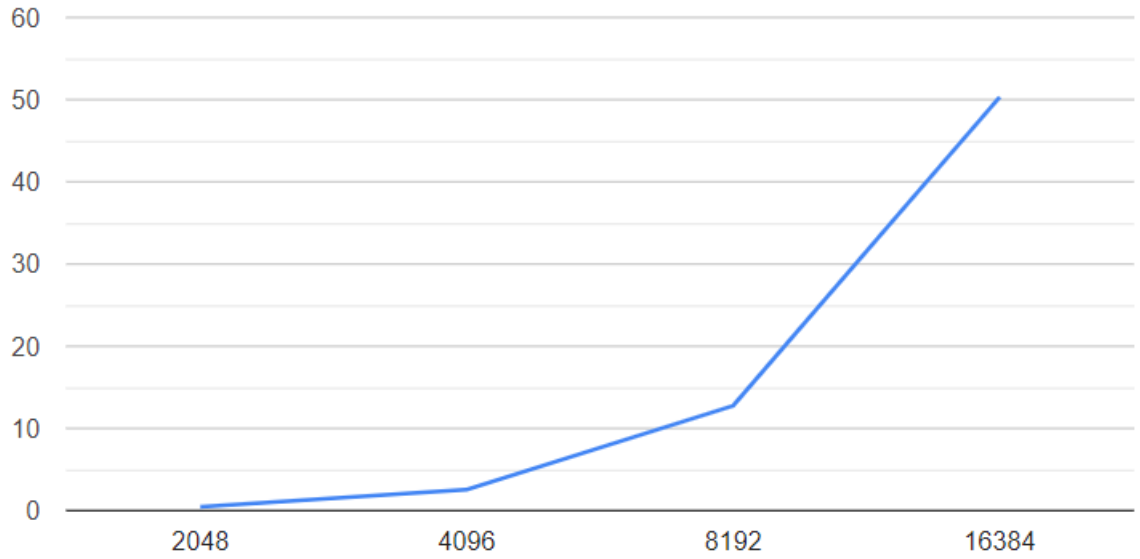

Рисунок 4.1 Час 1 ітерації алгоритму для різних розмірів популяції

Зі збільшенням розмірів генів в 4 рази, час 1 ітерації не збільшився навіть в 2 рази (Рисунок 4.2), тому що обчислення, які над ними проводяться є дуже простими.

Зі збільшенням розміру тестових даних, які використовуються в фітнес-функції для оцінки придатності генів, час зростає лінійно, що є досить непогано для такої складної функції.

Отже, на швидкість розробленого алгоритму істотно впливає тільки розмір популяції (Рисунок 4.3). Якщо використати алгоритм схрещення без найповільнішого алгоритму сортування, тоді його швидкість роботи буде дуже високою.

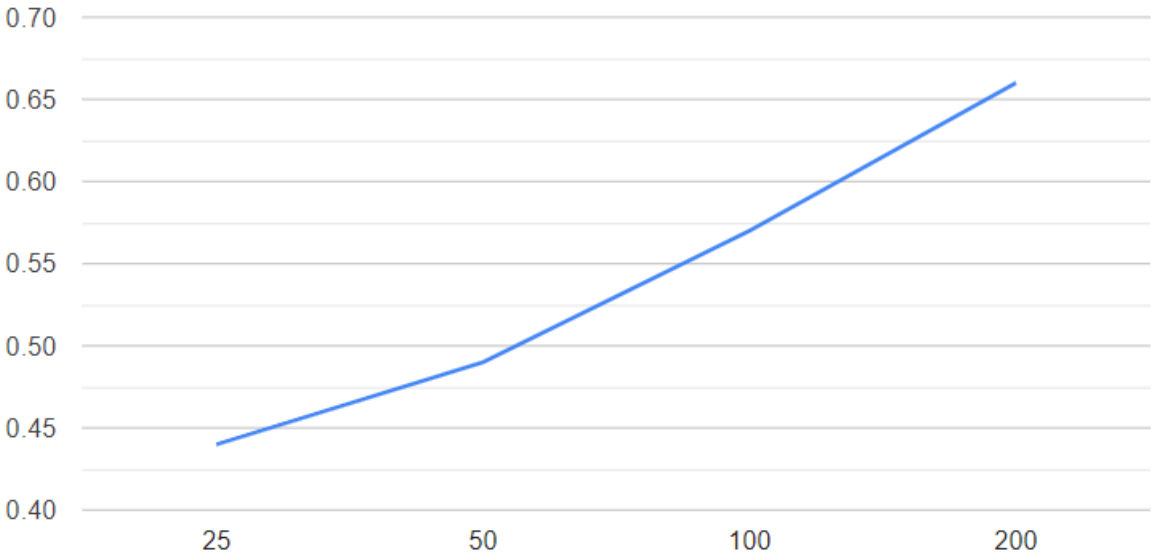

Рисунок 4.2 Час 1 ітерації алгоритму для різних розмірів генів

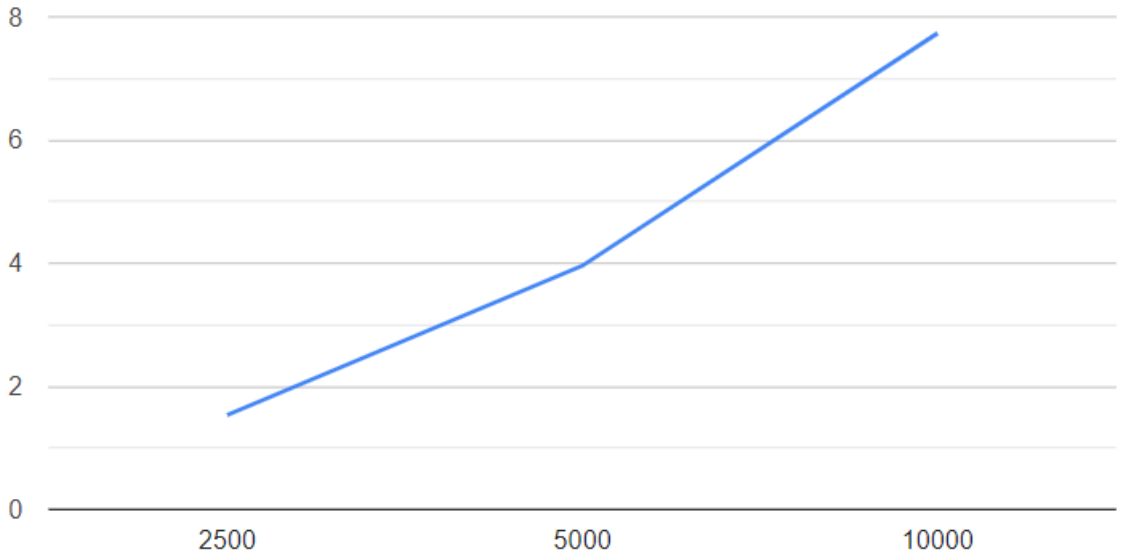

Рисунок 4.3 Час ітерації алгоритму для різних розмірів тестових даних

5. Висновки

Розроблений еволюційний алгоритм ранжування з використанням архітектури Nvidia CUDA може використовуватись в реальних пошукових системах, тому що:

- Багато обчислень виконується паралельно
- Відсутні жорсткі прив’язки по вхідним параметрам
- Оптимальна робота з пам’яттю
- З реальними тестовими наборами має забезпечити результати близькі до найкращих
- Є можливість модифікувати кожен етап алгоритму
- Кількість генів та фітнес функція не сильно сповільнюють роботу алгоритму.

Основною проблемою даного алгоритму є дуже повільний алгоритм схрещення, покращивши який, можна істотно збільшити його швидкість. Не менш важливим є те, що алгоритм залежить від якості вхідних даних, за допомогою яких виконується оцінка пристосованості популяції. Також алгоритм не гарантує знаходження найкращого розв’язку, але дозволяє отримати розв’язки близькі до найкращих.

Список використаної літератури

1. The Anatomy of a Large-Scale Hypertextual Web Search Engine : веб-сайт. URL: http://infolab.stanford.edu/~backrub/google.html

2. Google PageRank жив: почему он всё ещё важен : веб-сайт. URL: https://ahrefs.com/blog/ru/google-pagerank/

3. Google's 200 Ranking Factors: The Complete List (2020) : веб-сайт. URL: https://backlinko.com/google-ranking-factors

4. Introduction to GPUs CUDA : веб-сайт. URL:

https://nyu-cds.github.io/python-gpu/02-cuda/

5. CUDA C++ Programming Guide : веб-сайт. URL: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

6. CUDA C++ Best Practices Guide : веб-сайт. URL: https://docs.nvidia.com/cuda/cuda-c-best-practices-guide/index.html

7. Darrel Whitley *A Genetic Algorithm Tutorial*. 38 с. URL: https://www.cs.colostate.edu/~genitor/MiscPubs/tutorial.pdf

8. Real-coded genetic algorithms : веб-сайт. URL:

http://masters.donntu.org/2006/kita/bashev/library/rcga.html