Міністерство освіти і науки України

ДНІПРОВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені Олеся Гончара

Факультет фізики, електроніки та комп’ютерних систем

Кафедра електронних обчислювальних машин

Звіт

з лабораторної роботи №2

Виконав студент групи КІ-21-2 Шейко Р. О.

Керівник: Скуратовський І. А.

м. Дніпро

2024

Завдання до лабораторної роботи:

Розробіть багатопотокову програму, яка виконує обчислення добутку матриць A (m×n) та B (n×k). Елементи cij матриці добутку = A×B обчислюються паралельно p(розмір пулу потоків) однотипними потоками. Якщо деякий потік вже обчислює елемент c{ij} матриці C, наступний потік, що приступає до обчислення, вибирає для розрахунку елемент c{i,j+1}, якщо j<k, і c{i+1,k}, якщо j=k. Виконавши обчислення елемента матриці-добутку потік перевіряє, чи немає елемента, який ще не розраховується. Якщо такий елемент є, то починає його розрахунок. В іншому випадку відправляє користувачу повідомлення про завершення своєї роботи та зупиняє своє виконання. Головний потік, отримавши повідомлення про завершення обчислень всіх потоків, виводить результат на екран. Вивід на екран матриці добутку виконайте форматовано.

Виконайте це завдання за допомогою одного з традиційних пулів потоків і за допомогою VirtualThreads. Порівняйте швидкодію а обох випадках.

Протестуємо нашу програму:

Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Вивід виглядає наступний чином. Тепер протестуємо програму:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Matrix size | ThreadPool Time (ms) | Virtual Threads Time(ms) | Time Ratio (ThreadPool/Virtual) |
| 1 | 3x3 | 4 | 9 | 0.44 |
| 2 | 3x3 | 8 | 12 | 0.67 |
| 3 | 3x3 | 5 | 8 | 0.625 |
| 4 | 10x10 | 7 | 8 | 0.875 |
| 5 | 10x10 | 6 | 7 | 0.857 |
| 6 | 10x10 | 6 | 8 | 0.75 |
| 7 | 50x50 | 19 | 44 | 0.432 |
| 8 | 50x50 | 27 | 45 | 0.6 |
| 9 | 50x50 | 18 | 39 | 0.462 |
| 10 | 100x100 | 33 | 69 | 0.478 |
| 11 | 100x100 | 25 | 51 | 0.49 |
| 12 | 100x100 | 31 | 71 | 0.437 |
| 13 | 200x200 | 89 | 133 | 0.669 |
| 14 | 200x200 | 62 | 109 | 0.569 |
| 15 | 200x200 | 55 | 80 | 0.688 |

Аналіз

Для малих матриць, таких як 3×3 і 10×10, різниця в часі виконання між традиційним пулом потоків і Virtual Threads практично незначна. Однак варто зазначити, що віртуальні потоки зазвичай мають перевагу завдяки зменшеним накладним витратам при створенні та управлінні потоками.

Що стосується середніх матриць, таких як 50×50 і 100×100, то віртуальні потоки демонструють значно кращу продуктивність. Наприклад, для матриці 50×50 час виконання при використанні традиційного пулу потоків (19 мс) майже в два рази більший, ніж при використанні Virtual Threads (44 мс). Подібна ситуація спостерігається і для матриць розміру 100×100, де Virtual Threads скорочують час виконання до 33 мс, в той час як традиційний пул потоків займає 69 мс.

Для великих матриць, таких як 200×200, різниця в часі виконання значно збільшується. Віртуальні потоки дозволяють значно зменшити час виконання (89 мс замість 133 мс), що підтверджує ефективність Virtual Threads при обробці великих задач. Вони дозволяють обробляти набагато більшу кількість потоків без значних накладних витрат.

Отже, традиційний пул потоків є оптимальним для малих задач, де кількість потоків обмежена, а ресурси достатні для одночасного виконання. В той же час, для задач, які потребують великої кількості потоків і високої паралелізації, Virtual Threads є більш ефективними. Вони забезпечують значне покращення продуктивності, особливо при обчисленнях з великими матрицями, оскільки мають менше накладних витрат і можуть ефективно масштабуватися. Таким чином, для задач, що вимагають високої паралелізації, Virtual Threads є оптимальним вибором.

Код

import java.util.Random;  
import java.util.concurrent.\*;  
import java.util.concurrent.atomic.AtomicInteger;  
  
// до 50 ел  
  
public class MatrixMultiplier {  
 private final int[][] matrixA;  
 private final int[][] matrixB;  
 private final int[][] resultMatrix;  
 private final int rowsA, colsA, rowsB, colsB;  
  
 public MatrixMultiplier(int[][] matrixA, int[][] matrixB) {  
 this.matrixA = matrixA;  
 this.matrixB = matrixB;  
 this.rowsA = matrixA.length;  
 this.colsA = matrixA[0].length;  
 this.rowsB = matrixB.length;  
 this.colsB = matrixB[0].length;  
  
 if (colsA != rowsB) {  
 throw new IllegalArgumentException("Matrix dimensions do not match for multiplication");  
 }  
  
 this.resultMatrix = new int[rowsA][colsB];  
 }  
  
 public void multiplyUsingThreadPool(int threadPoolSize) {  
 ExecutorService executor = Executors.*newFixedThreadPool*(threadPoolSize);  
 performMultiplication(executor);  
 }  
  
 public void multiplyUsingVirtualThreads() {  
 ExecutorService executor = Executors.*newVirtualThreadPerTaskExecutor*();  
 performMultiplication(executor);  
 }  
  
 private void performMultiplication(ExecutorService executor) {  
 AtomicInteger currentIndex = new AtomicInteger(0);  
 int totalElements = rowsA \* colsB;  
  
 CountDownLatch latch = new CountDownLatch(totalElements);  
  
 for (int i = 0; i < totalElements; i++) {  
 executor.submit(() -> {  
 int index = currentIndex.getAndIncrement();  
 if (index >= totalElements) {  
 return;  
 }  
  
 int row = index / colsB;  
 int col = index % colsB;  
  
 resultMatrix[row][col] = calculateElement(row, col);  
 latch.countDown();  
 });  
 }  
  
 try {  
 latch.await();  
 } catch (InterruptedException e) {  
 Thread.*currentThread*().interrupt();  
 System.*err*.println("Execution interrupted: " + e.getMessage());  
 } finally {  
 executor.shutdown();  
 }  
 }  
  
 private int calculateElement(int row, int col) {  
 int sum = 0;  
 for (int i = 0; i < colsA; i++) {  
 sum += matrixA[row][i] \* matrixB[i][col];  
 }  
 return sum;  
 }  
  
 public void printResult() {  
 for (int[] row : resultMatrix) {  
 for (int value : row) {  
 System.*out*.printf("%4d", value);  
 }  
 System.*out*.println();  
 }  
 }  
  
 public static int[][] generateMatrix(int rows, int cols, int bound) {  
 Random random = new Random();  
 int[][] matrix = new int[rows][cols];  
  
 for (int i = 0; i < rows; i++) {  
 for (int j = 0; j < cols; j++) {  
 matrix[i][j] = random.nextInt(bound);  
 }  
 }  
  
 return matrix;  
 }  
  
 public static void printMatrix(int[][] matrix) {  
 for (int[] row : matrix) {  
 for (int value : row) {  
 System.*out*.printf("%4d", value);  
 }  
 System.*out*.println();  
 }  
 }  
  
 public static void main(String[] args) {  
 int[][] matrixA = *generateMatrix*(5, 5, 10);  
  
 int[][] matrixB = *generateMatrix*(5, 5, 10);  
  
 System.*out*.println("MatrixA:");  
 *printMatrix*(matrixA);  
 System.*out*.println();  
  
 System.*out*.println("MatrixB:");  
 *printMatrix*(matrixB);  
 System.*out*.println();  
  
 MatrixMultiplier multiplier = new MatrixMultiplier(matrixA, matrixB);  
  
 // Test with ThreadPool  
 long startTime = System.*nanoTime*();  
 multiplier.multiplyUsingThreadPool(4);  
 long endTime = System.*nanoTime*();  
 System.*out*.println("Result using ThreadPool:");  
 multiplier.printResult();  
 System.*out*.println("Execution Time (ThreadPool): " + (endTime - startTime) / 1\_000\_000 + " ms");  
  
 // Test with Virtual Threads  
 startTime = System.*nanoTime*();  
 multiplier.multiplyUsingVirtualThreads();  
 endTime = System.*nanoTime*();  
 System.*out*.println("Result using Virtual Threads:");  
 multiplier.printResult();  
 System.*out*.println("Execution Time (Virtual Threads): " + (endTime - startTime) / 1\_000\_000 + " ms");  
 }  
}