Университет ИТМО

Институт прикладных компьютерных наук
Глубокое обучение и генеративный искусственный интеллект

ОТЧЕТ ПО 1-Й ЛАБОРАТОРНОЙ РАБОТЕ
курса
«Эволюционные вычисления»
Вариант 4

СЛОЖНОСТЬ АЛГОРИТМОВ И ИХ ОПТИМИЗАЦИЯ

Студент:
Группа № M4130 *Батурина Ксения Александровна*

Санкт-Петербург 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Для реализации был выбран ЯП **Python**.
Код доступен в репозитории на GitHub: https://github.com/xeniabaturina/ITMO_EVOL/tree/main/lab1.

Цель работы:

Получить навыки вычисления сложности алгоритмов и их оптимизации различными методами.

Задачи работы:

1. Реализовать на любом ЯП алгоритм, согласно варианту задания.
2. Вычислить сложность алгоритма, привести расчёты, результаты нагрузочных тестов с замером затраченного времени и ресурсов.
3. Выполнить оптимизацию как алгоритмическую если возможно, с выносом инварианта, например, так и программными методами выбранного ЯП.
4. Вычислить сложность оптимизированного алгоритма, привести расчёты, результаты нагрузочных тестов с замером затраченного времени и ресурсов.
5. Описать различие величин сложности, результатов, привести обоснование.
6. Сформулировать выводы.
7. Приложить код в виде ссылки на публичный репозиторий.

1 MERGE SORT

1.1 Определение

Сортировка слиянием (`Merge Sort`) — это эффективный метод упорядочивания элементов в массиве. Он основан на принципе «разделяй и властвуй». Процесс сортировки слиянием включает в себя несколько шагов:

- Разделение (`Splitting`): Исходный массив разделяется на две половины. Этот шаг продолжается до тех пор, пока каждая подмассив достигнет минимальной длины.
- Сортировка (`Sorting`): Каждая подмассив сортируется индивидуально, обычно с использованием того же метода сортировки слиянием. Этот шаг гарантирует, что каждая часть массива отсортирована.
- Слияние (`Merging`): Отсортированные подмассивы объединяются обратно в один массив. Происходит сравнение элементов на каждом этапе, и элементы вставляются в результирующий массив в правильном порядке.

1.2 Реализация

Ниже приведен листинг с кодом реализованной сортировки слиянием:

```
def merge_sort(arr):
    # Check if the array has more than one element
    if len(arr) > 1:
        # Calculate the midpoint of the array
        mid = len(arr) // 2

        # Divide the array into two halves
        left_half = arr[:mid]
        right_half = arr[mid:]

        # Recursively apply merge_sort to both halves
        merge_sort(left_half)
        merge_sort(right_half)
```

```
    # Initialize indices for left, right, and main arrays
    i = j = k = 0

    # Merge the sorted halves back into the main array
    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            arr[k] = left_half[i]
            i += 1
        else:
            arr[k] = right_half[j]
            j += 1
        k += 1

    # If there are remaining elements in left_half,
    # add them to the main array
    while i < len(left_half):
        arr[k] = left_half[i]
        i += 1
        k += 1

    # If there are remaining elements in right_half,
    # add them to the main array
    while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1
```

1.3 Вычислительная сложность

Сортировка слиянием имеет вычислительную сложность $O(n \log n)$, где n — количество элементов в сортируемом массиве. Это означает, что время выполнения алгоритма растет логарифмически пропорционально увеличению размера входных данных.

Пояснение:

На каждом уровне рекурсии массив делится на две части. Это деление происходит $log\, n$ раз, где $logn$ — логарифм по основанию 2 от числа элементов в массиве. Каждый уровень разделения занимает $O(1)$ времени.

При этом выполняется слияние, которое также происходит $log\, n$ раз. Каждый элемент массива участвует в не более чем двух слияниях. Поэтому слияние на каждом уровне занимает $O(n)$ времени, где n — общее количество элементов в массиве.

Общая сложность равна количеству уровней разделения ($log\, n$) умноженному на время слияния на каждом уровне ($O(n)$). Таким образом, общая сложность сортировки слиянием — $O(n\, log\, n)$.

2 ОПТИМИЗАЦИЯ

2.1 Алгоритмическая оптимизация

2.1.1 Hybrid Merge Sort

Гибридный алгоритм сортировки `Hybrid Merge Sort` — оптимизация алгоритма `Merge Sort`, которая объединяет идеи сортировки слиянием и сортировки вставками (`Insertion Sort`). Основное отличие от обычной сортировки слиянием заключается в том, что для небольших подмассивов используется сортировка вставками вместо рекурсивного вызова сортировки слиянием. Это позволяет уменьшить количество рекурсивных вызовов и, таким образом, улучшить производительность для маленьких массивов.

Код реализации гибридного алгоритма сортировки приведен в листинге:

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        # Shift elements greater than key to the right
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1

        # Insert the key in its correct position
        arr[j + 1] = key


def hybrid_merge_sort(arr, threshold=10):
    if len(arr) <= threshold:
        # Use insertion sort for small subarrays
        insertion_sort(arr)
    else:
        # Continue with merge sort for larger subarrays
        mid = len(arr) // 2
        left_half = arr[:mid]
```

```
    right_half = arr[mid:]

    hybrid_merge_sort(left_half, threshold)
    hybrid_merge_sort(right_half, threshold)

    i = j = k = 0

    # Merge the sorted subarrays
    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            arr[k] = left_half[i]
            i += 1
        else:
            arr[k] = right_half[j]
            j += 1
        k += 1

    # If there are remaining elements in left_half,
    # add them to the main array
    while i < len(left_half):
        arr[k] = left_half[i]
        i += 1
        k += 1

    # If there are remaining elements in right_half,
    # add them to the main array
    while j < len(right_half):
        arr[k] = right_half[j]
        j += 1
        k += 1
```

Вычислительная сложность

Алгоритм Hybrid Merge Sort включает в себя:

- Сортировку вставками: В худшем случае имеет квадратичную сложность $O(n^2)$, где n — количество элементов в подмассиве.
- Сортировку слиянием: вычислительная сложность та же, что и у обычной сортировки слиянием — $O(n \log n)$, где n — общее количество элементов в массиве.

Гибридный алгоритм сортировки оценивает размер каждого подмассива перед рекурсивным вызовом и применяет сортировку вставками, если размер подмассива не превышает заданный порог. Таким образом, алгоритм более эффективен для маленьких массивов, что улучшает общую производительность сортировки слиянием.

2.2 Оптимизация программными методами языка Python

2.2.1 Numba

Numba — это компилятор для Python, который может автоматически компилировать функции Python в машинный код для улучшения их производительности. Numba обеспечивает значительный прирост производительности для определенных типов задач.

Основные характеристики Numba:

- JIT–компиляция (Just-In-Time): Numba использует JIT-компиляцию, что означает, что функции Python компилируются в машинный код во время выполнения программы. Это позволяет избежать интерпретации Python и значительно ускоряет выполнение кода.
- Поддержка NumPy-подобных массивов: Numba интегрирован с библиотекой NumPy и спроектирован для эффективной обработки многомерных массивов данных.
- Статическая и динамическая типизация: Numba позволяет явно указывать типы данных для функций, что может улучшить производительность, особенно при работе с числовыми данными. Кроме того, Numba также поддерживает динамическую типизацию.

Для оптимизации кода с помощью Numba необходимо импортировать библиотеку Numba: `import numba`. Декоратор `@jit` указывает, что функцию следует скомпилировать в машинный код с использованием Numba.

Merge Sort + Numba

Оптимизация Merge Sort с помощью Numba:

```
@numba.njit
def merge_sort_numba(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = merge_sort_numba(arr[:mid])
    right_half = merge_sort_numba(arr[mid:])

    return merge_numba(left_half, right_half)

@numba.njit
def merge_numba(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])

    return result
```

Hybrid Merge Sort + Numba

Оптимизация Hybrid Merge Sort с помощью Numba:

```
@numba.njit
def insertion_sort_numba(arr):
    for i in range(1, len(arr)):
        key = arr[i]
```

```
        j = i - 1
        while j >= 0 and key < arr[j]:
            arr[j + 1] = arr[j]
            j -= 1
        arr[j + 1] = key

@numba.njit
def hybrid_merge_sort_numba(arr, threshold=10):
    if len(arr) <= threshold:
        insertion_sort_numba(arr)
    else:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        hybrid_merge_sort_numba(left_half, threshold)
        hybrid_merge_sort_numba(right_half, threshold)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

3 РЕЗУЛЬТАТЫ ТЕСТИРОВАНИЯ И СРАВНЕНИЕ

3.1 Тестирование

Было проведено тестирование всех реализованных решений:

- Merge Sort,
- Hybrid Merge Sort,
- Merge Sort (Optimized via Numba),
- Hybrid Merge Sort (Optimized via Numba).

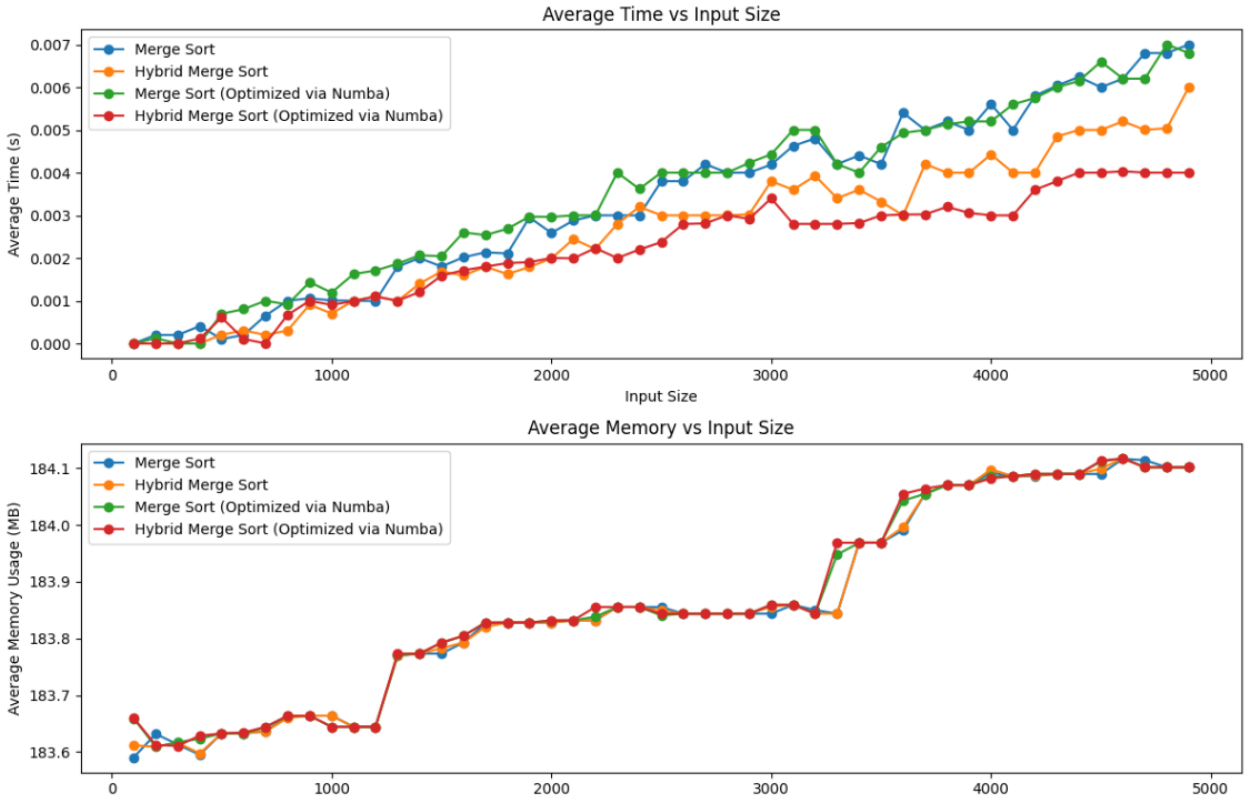
Методология тестирования:

- Для каждого алгоритма проведено пять независимых нагрузочных тестов.
- Размер входных данных варьировался от 100 элементов до 4900 элементов с шагом 100.
- Замерялось среднее время выполнения и среднее использование памяти для каждого теста.
- Все тесты были проведены на случайно сгенерированных массивах.

3.2 Результаты тестирования

На Рисунке 1 приведены результаты проведенного тестирования, показывающие зависимость затраченных времени и ресурсов от размера входного массива.

Рисунок 1 — Результаты нагрузочных тестов с замером затраченных времени и памяти



4 ВЫВОДЫ

4.1 Время выполнения

При небольших объемах данных сортировка слиянием и гибридная сортировка слиянием демонстрируют схожую производительность, однако чем выше размерность задачи, тем сильнее заметна разница — гибридная сортировка слиянием демонстрирует более высокую скорость. Кроме того, оптимизированные версии с использованием Numba показывают заметное ускорение в сравнении с базовыми реализациями, особенно при увеличении размера входных данных.

4.2 Использование памяти

И сортировка слиянием, и гибридная сортировка слиянием обладают схожими характеристиками по использованию памяти.

4.3 Общий вывод

Оптимизированные версии алгоритмов с использованием Numba предоставляют значительный выигрыш в производительности, делая их хорошим выбором для обработки больших объемов данных. Гибридная сортировка слиянием, особенно оптимизированная, показывает налучший результат.