Національний технічний університет України

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра Обчислювальної Техніки

Лабораторна робота №1

з дисципліни "Проектування складних систем"

Тема: " Дослідження протоколу когерентності кешів MESIF"

Виконав:

студент групи ІП-93

Домінський В.О.

Перевірив:

Долголенко О. М.

Київ 2023

Зміст

Мета: 3
Вихідні дані: 3
Хід роботи 4
Висновок: 11
Посилання: 12

Мета:

Розробіть два екземпляра свого варіанта багатопотокової програми на Java. Перший екземпляр багатопотокової програми розробіть за додержанням принципу локалізації, при цьому як скалярні так і векторні змінні не повинні розділюватися між потоками – механізм когерентності кешів MESIF при цьому працювати не буде. Другий екземпляр багатопотокової програми розробіть за використанням скалярних та векторних глобальних змінних та механізмів синхронізації потоків при зверненні до цих змінних – пошук актуального значення змінної при цьому буде відбуватися, відповідно до механізму когерентності кешів MESIF, у кешах L1D всіх ядер мікропроцесора. Побудуйте порівняльні графіки й зробіть висновки по відносній швидкодії цих екземплярів програми

Вихідні дані:

Варіант – 12

Функції:

- A=B*MC+D*MZ+E*MM
- MG=min(D+E)*MM*MT-MZ*ME

Хід роботи

Для початку Ми створюємо Наші власні конструкції для вектору та матриці:

```
class LabMathConstructs:
  def vector(self, d):
    return np.array(
        [
            round(
                uniform(MIN_FLOAT_VALUE, MAX_FLOAT_VALUE),
                randint(MIN_FLOAT_EXPONENT, MAX_FLOAT_EXPONENT)
            )
            for _ in range(d)
          ]
    )

  def matrix(self, d):
    matrix = np.array(
        [
            round(
                uniform(MIN_FLOAT_VALUE, MAX_FLOAT_VALUE),
                randint(MIN_FLOAT_EXPONENT, MAX_FLOAT_EXPONENT)
            )
            for _ in range(d ** 2)
        ]
    )
    return matrix.reshape((d, d))
```

Наступний крок – записати їх до якогось місця зберігання. У даному випадку – excel файл:

```
mc = LabMathConstructs()

vector_b = [mc.vector(d) for d in DIM_RANGE]
vector_d = [mc.vector(d) for d in DIM_RANGE]
vector_e = [mc.vector(d) for d in DIM_RANGE]
matrix_mc = [mc.matrix(d) for d in DIM_RANGE]
matrix_mz = [mc.matrix(d) for d in DIM_RANGE]
matrix_mm = [mc.matrix(d) for d in DIM_RANGE]
matrix_mt = [mc.matrix(d) for d in DIM_RANGE]
matrix_me = [mc.matrix(d) for d in DIM_RANGE]

df = pd.DataFrame(list(zip(vector_b, vector_d, vector_e, matrix_mc,
matrix_mz, matrix_mm, matrix_mt, matrix_me)), columns=DATA_COLUMNS)
df.to_excel(EXCEL_NAME)
df.head()
```

Вивід в Google colab:

	vector_b	vector_d	vector_e	matrix_mc	matrix_mz	matrix_mm	matrix_mt	matrix_me
0	[24.357, 20.14, 5.506, 19.54, 11.935, 22.874, ...	[15.42, 23.56, 12.447, 4.0688, 3.97, 19.446, 8...	[22.03, 15.799, 0.3968, 21.242, 21.31, 21.17, ...	[[16.358, 17.146, 17.9687, 17.394, 19.584, 7.5...	[[24.2088, 0.5718, 9.251, 1.252, 11.8832, 2.64...	[[7.99, 17.81, 20.23, 11.98, 0.2167, 21.764, 1...	[[4.58, 23.3604, 12.941, 11.12, 16.3189, 3.97,...	[[20.98, 20.84, 0.64, 12.3698, 24.57, 13.849, ...
1	[16.011, 3.7, 0.119, 9.567, 20.981, 22.7342, 4...	[22.5948, 19.2631, 4.22, 7.5987, 1.7534, 11.62...	[12.3095, 19.53, 9.3953, 24.59, 14.7455, 4.155...	[[3.7294, 2.9, 15.5082, 19.235, 20.49, 14.85, ...	[[6.38, 11.5022, 8.86, 13.42, 14.775, 6.6838, ...	[[13.0, 6.3715, 17.96, 4.13, 2.0932, 24.1272, ...	[[4.83, 12.6917, 10.5489, 18.3203, 16.865, 15....	[[15.9474, 19.7, 2.638, 6.48, 13.6257, 0.5658,...
2	[6.55, 24.36, 11.894, 15.42, 2.185, 8.37, 18.7...	[5.746, 6.866, 18.732, 6.75, 9.2424, 9.091, 6....	[8.2481, 19.314, 15.1781, 15.0, 24.502, 18.717...	[[6.667, 16.95, 24.65, 6.7648, 6.922, 13.7427,...	[[21.548, 13.9488, 1.4547, 2.76, 10.813, 9.483...	[[24.2868, 1.67, 5.81, 6.0, 19.919, 22.069, 18...	[[19.659, 10.7354, 17.991, 22.41, 20.875, 4.19...	[[13.1836, 18.42, 0.6845, 14.53, 19.079, 24.86...
3	[12.97, 21.16, 0.813, 20.7159, 5.736, 14.914, ...	[13.07, 13.47, 14.088, 17.89, 6.9183, 0.558, 2...	[10.69, 15.4222, 12.1678, 10.49, 7.4913, 11.49...	[[11.36, 3.519, 24.4171, 16.03, 21.0309, 19.35...	[[17.956, 11.7466, 3.537, 3.6158, 12.078, 12.1...	[[6.95, 9.2262, 9.127, 22.3464, 15.9733, 21.1,...	[[17.5588, 11.96, 9.986, 16.504, 6.1164, 11.07...	[[24.84, 7.1, 18.69, 20.5755, 18.594, 20.612, ...
4	[7.55, 7.67, 12.0061, 7.1059, 8.9345, 13.295, ...	[4.833, 15.19, 20.291, 8.3204, 6.7634, 7.6204,...	[10.16, 5.1, 19.35, 3.51, 8.4718, 1.0536, 22.3...	[[8.454, 8.507, 7.875, 17.542, 6.66, 17.4915, ...	[[2.6944, 3.349, 16.84, 10.325, 23.08, 8.04, 7...	[[12.698, 1.3776, 8.81, 17.5755, 15.63, 22.309...	[[22.965, 11.6743, 13.7979, 18.553, 23.45, 7.0...	[[22.2889, 14.466, 12.8825, 24.4342, 19.3829, ...

Але ж треба ще кудись виписувати результати. Для цього варіанту будемо використовувати звичайний .txt формат. Спочатку потрібно перевірити чи файл уже існує: якщо так, то його треба видалити:

```
if (os.path.exists(RESULT_FILE)):
  os.remove(RESULT_FILE)

def save_result_as_file(text):
  file = open(RESULT_FILE, "a")
  file.write(f"{text}")
  file.close()
```

Також потрібно створити функції для обрахування Наших виразів. Оформити їх Я вирішив у вигляді 2-ох класів для легшого тестування обчислень в різних потоках - у кожному з них є метод, котрий відповідає за власний функціонал: хтось множить 2 матриці, хтось – вектор на матрицю і т.д.:

```
class FirstStatement:
  def b_dot_mc(self, b, mc):
    return np.dot(b, mc)

  def d_dot_mz(self, d, mz):
    return np.dot(d, mz)

  def e_dot_mm(self, e, mm):
    return np.dot(e, mm)

  def result_a(self, b_dot_mc, d_dot_mz, e_dot_mm):
    res = np.add(b_dot_mc, np.add(d_dot_mz, e_dot_mm))
    save_result_as_file(res)
    print(res)
```

```python
class SecondStatement:
  def min_d_add_e(self, d, e):
    return np.amin(np.add(d, e))

  def mm_dot_mt(self, mm, mt):
    return np.dot(mm, mt)

  def mz_dot_me(self, mz, me):
    return np.dot(mz, me)

  def result_mg(self, min_d_add_e, mm_dot_mt, mz_dot_me):
    res = np.subtract(np.dot(min_d_add_e, mm_dot_mt), mz_dot_me)
    save_result_as_file(res)
    print(res)
```

Подальші класи будуть використовувати функціонал вище і записуватимуть результати до глобальних змінних, аби їх бачили всі потоки

```python
class LabFirstThread:
    def __init__(self):
      self.fs = FirstStatement()
      self.first_thread_result = None
      self.second_thread_result = None
      self.third_thread_result = None

    def first_thread(self, b, mc):
        self.first_thread_result = self.fs.b_dot_mc(b, mc)

    def second_thread(self, d, mz):
        self.second_thread_result = self.fs.d_dot_mz(d, mz)

    def third_thread(self, e, mm):
        self.third_thread_result = self.fs.e_dot_mm(e, mm)

    def fourth_thread(self):
        if self.first_thread_result.all() and self.second_thread_result.all() and
self.third_thread_result.all():
            self.fs.result_a(self.first_thread_result, self.second_thread_result,
self.third_thread_result)
```

```
class LabSecondThread:
    def __init__(self):
      self.ss = SecondStatement()
      self.first_thread_result = None
      self.second_thread_result = None
      self.third_thread_result = None

    def first_thread(self, d, e):
        self.first_thread_result = self.ss.min_d_add_e(d, e)

    def second_thread(self, mm, mt):
        self.second_thread_result = self.ss.mm_dot_mt(mm, mt)

    def third_thread(self, mz, me):
        self.third_thread_result = self.ss.mz_dot_me(mz, me)

    def fourth_thread(self):
        if self.first_thread_result.all() and self.second_thread_result.all() and
self.third_thread_result.all():
            self.ss.result_mg(self.first_thread_result, self.second_thread_result,
self.third_thread_result)
```

Тепер найголовніший та найбільший клас усього проекту – LabRuns.

Його ціль - вираховування швидкості обчислень у потоках, який має такі елементи:

- Масиви точок для будування графіків
- Початок та кінець відліку часу та виведення до консолі часу виконання
- Методи first_run та second_run, котрі запускають обчислення виразів та часу на їх обробку
- run_first_thread_first_run та run_second_thread_first_run – для вираховування виразів в одному потоці
- run_first_threads_first_run та run_second_threads_first_run, котрі компонують кілька потоків з різними частинами рівнянь

```
class LabRuns:
  def __init__(self):
    self.xs_1_1 = []
    self.ys_1_1 = []

    self.xs_1_2 = []
    self.ys_1_2 = []

    self.xs_2_1 = []
    self.ys_2_1 = []

    self.xs_2_2 = []
    self.ys_2_2 = []

    self.starting_time = 0
    self.ending_time = 0
    self.actual_time = 0

  def start_time(self, thread_num, run_num):
      self.starting_time = time.time()
      self.print_thread_start(thread_num, run_num)

  def end_time(self):
      self.ending_time = time.time()
      self.actual_time = self.ending_time - self.starting_time

  def print_thread_start(self, thread_num, run_num):
    print(f"Start Thread #{thread_num}_{run_num}")

  def run_first_thread_first_run(self, i):
    fs = FirstStatement()

    thread = Thread(target=fs.result_a, args=[fs.b_dot_mc(vector_b[i], matrix_mc[i]),
fs.d_dot_mz(vector_d[i], matrix_mz[i]), fs.e_dot_mm(vector_e[i], matrix_mm[i]),],)
    thread.start()
    thread.join()

  def run_second_thread_first_run(self, i):
    ss = SecondStatement()

    thread = Thread(target=ss.result_mg, args=[ss.min_d_add_e(vector_d[i], vector_e[i]),
ss.mm_dot_mt(matrix_mm[i], matrix_mt[i]), ss.mz_dot_me(matrix_mz[i], matrix_me[i]),],)
    thread.start()
    thread.join()

  def first_run(self):
    for i in range(0, len(DIM_RANGE)):
        self.ys_1_1.append(len(vector_b[i]))

        self.start_time(i + 1, 1)
        self.run_first_thread_first_run(i)
        self.end_time()

        self.xs_1_1.append(self.actual_time)
        self.ys_1_2.append(len(vector_b[i]))

        self.start_time(i + 1, 2)
        self.run_second_thread_first_run(i)
        self.end_time()

        self.xs_1_2.append(self.actual_time)
```

```
def run_first_threads_second_run(self, i):
        self.ys_2_1.append(len(vector_b[i]))

        thread = LabFirstThread()

        first_threads = []

        self.start_time(i + 1, 1)

        first_threads.append(Thread(target=thread.first_thread, args=[vector_b[i],
matrix_mc[i]],))
        first_threads.append(Thread(target=thread.second_thread, args=[vector_d[i],
matrix_mz[i]],))
        first_threads.append(Thread(target=thread.third_thread, args=[vector_e[i],
matrix_mm[i]],))
        first_threads.append(Thread(target=thread.fourth_thread))

        for thread in first_threads:
          thread.start()
        for thread in first_threads:
          thread.join()

        self.end_time()

        self.xs_2_1.append(self.actual_time)
        self.ys_2_2.append(len(vector_b[i]))

  def run_second_threads_second_run(self, i):
        thread = LabSecondThread()
        second_threads = []

        self.start_time(i + 1, 2)

        second_threads.append(Thread(target=thread.first_thread, args=[vector_d[i],
vector_e[i]],))
        second_threads.append(Thread(target=thread.second_thread, args=[matrix_mm[i],
matrix_mt[i]],))
        second_threads.append(Thread(target=thread.third_thread, args=[matrix_mz[i],
matrix_me[i]],))
        second_threads.append(Thread(target=thread.fourth_thread))

        for thread in second_threads:
          thread.start()
        for thread in second_threads:
          thread.join()

        self.end_time()
        self.xs_2_2.append(self.actual_time)

  def second_run(self):
    for i in range(0, len(DIM_RANGE)):
        self.run_first_threads_second_run(i)
        self.run_second_threads_second_run(i)
```

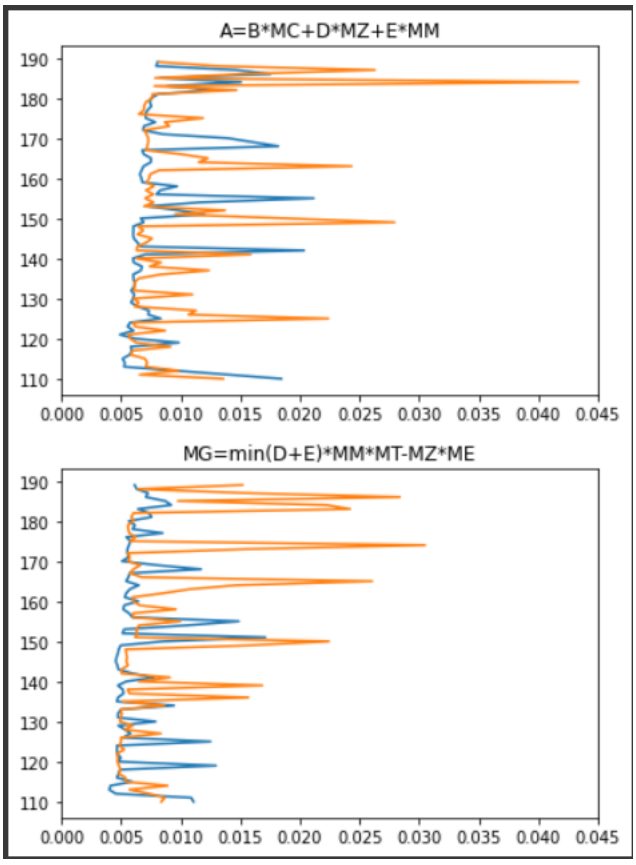
При запуску обох методів first_run та second_run Нам виводить ось таку інформацію:

```
runs = LabRuns()

[600] runs.first_run()
 -15367.17679311 -11020.13723205]
[-11204.71020731 -12359.60240436 -13115.59607952 ... -11716.28837944
 -16235.53630452 -13018.62107122]
...
[-12273.94940289 -12595.58379354 -11588.27140837 ... -13774.4137325
 -15207.57483641 -11889.01433919]
[ -8720.45304727  -9315.54627577 -11748.96587671 ... -10904.06743125
 -12947.96109243  -9129.54365474]
[-11927.40992055 -13196.65402522 -13254.03202602 ... -11150.82886786
 -15865.36710451 -12781.17196899]]
Start Thread #80_1
[89958.34995726 88098.15891119 92404.52898545 87754.94055619
 82584.31088727 87712.48183346 85515.87967704 89733.38285691
 83155.4943299  84942.58941969 81934.94317739 90305.07047744
```

Як видно, то Ми маємо результати виразів, та позначки для розуміння, коли починається новий потік

Ну і в самому кінці Нам треба відобразити показники Наших потоків:

```
plt.title(FIRST_PLOT)
plt.xlim(0, 0.045)
plt.plot(runs.xs_1_1, runs.ys_1_1)
plt.plot(runs.xs_2_1, runs.ys_2_1)
plt.show()

plt.title(SECOND_PLOT)
plt.xlim(0, 0.045)
plt.plot(runs.xs_1_2, runs.ys_1_2)
plt.plot(runs.xs_2_2, runs.ys_2_2)
plt.show()
```



Висновок:

На жаль пророблена робота не дала бажаних результатів, а саме пришвидшення обрахування всіх виразів, а навпаки – зробила тільки гірше. Можливо було не зовсім коректно проведено синхронізацію; можливо це пов’язано з несправедливим розподілом задач (деякі потоки роблять більш складні операції, ніж інші) - точно Я сказати в даному випадку не можу.

У ході виконання роботи Я створив 2 варіанти багатопотокової програми:

1. Перший – мав в собі принципи локалізації - при цьому як скалярні так і векторні змінні не повинні розділюватися між потоками – механізм когерентності кешів MESIF при цьому працювати не буде
2. У другий ж навпаки – було використано глобальні змінні для скалярів та матриць за допомогою механізмів синхронізації - пошук актуального значення змінної при цьому буде відбуватися, відповідно до механізму когерентності кешів MESIF, у кешах L1D всіх ядер мікропроцесора

Побудував кілька графіків, котрі порівнюють обидві варіації та зробив невтішні висновки щодо швидкодії багатопотокового методу.

Додатково створив запис значень в окремі файли – excel та txt

Посилання:

1. Робота на Google Colab – посилання
2. Робота на GitHub – посилання